Advantech DLL Drivers User's Manual and Programmer's Reference

1st Edition

Advantech Customer Services

Each and every Advantech product is built to the most exacting specifications to ensure reliable performance in the harsh and demanding conditions typical of industrial environments. Whether your new Advantech equipment is destined for the laboratory or the factory floor, you can be assured that your product will provide the reliability and ease of operation for which the name Advantech has come to be known.

Your satisfaction is our primary concern. Here is a guide to Advantech's customer services. To ensure you get the full benefit of our services, please follow the instructions below carefully.

Technical Support

We want you to get the maximum performance from your products. So if you run into technical difficulties, we are here to help. For the most frequently asked questions, you can easily find answers in your product documentation. These answers are normally a lot more detailed than the ones we can give over the phone.

So please consult this manual first. If you still cannot find the answer, gather all the information or questions that apply to your problem, and with the product close at hand, call your dealer. Our dealers are well trained and ready to give you the support you need to get the most from your Advantech products. In fact, most problems reported are minor and are able to be easily solved over the phone.

In addition, free technical support is available from Advantech engineers every business day. We are always ready to give advice on application requirements or specific information on the installation and operation of any of our products.

Technical Suppport Offices

USA	American Advantech Corporation 750 East Arques Avenue Sunnyvale, CA 94086 Tel: (408)245-6678 Fax: (408)245-5678 E-mail: IAInfo@advantech.com
Asia	Advantech Co., LTD 4th Floor, 108-3 Min-Chuan Road Shing-Tien City, Taipei County, Taiwan ROC Tel: (+886-2) 2218-4567 Fax: (+886-2) 2218-1989 E-mail: IASupport@advantech.com.tw
Europe	**Advantech Germany** Karlsruherstr. 11/1 D-70771 Leinf.-Echterdingen Germany Tel: +49 (0) 711 797 333 60 Fax: +49 (0) 711 797 333 85 **Advantech Italy** Via Don Verderio 4/B-20060 Cassina de, Pecchi (MI), Italy Tel: +39-2-95343054 Fax: +39-2-95343067
Mainland China	**Beijing office**: No. 7, 6th Street, Shang Di Zone Haidian District, 100085 Beijing, China Tel: +86-10-62984345~47, 62986314~17 Fax: +86-1-62984341~42 **Shanghai office**: Room #701, 7th Floor, Hua-Fu Building A 585 Long Hua W. Road 200232 Shanghai, China Tel: +86-21-64696831, 64697910 Fax: +86-21-64696834

Limited Warranty

Advantech Corporation does not warrant that the 32-bit DLL Drivers software package will function properly in every hardware/software environment. Advantech Corporation makes no representation or warranties of any kinds whatsoever with respect to the contents of this manual and specifically disclaims any implied warranties or fitness for any particular purpose. Advantech Corporation shall not be held liable for errors in this manual or for incidental or consequential damages in connection with the use of this manual or its contents. Advantech Corporation reserves the right to revise this manual at any time without prior notice.

About This Manual

This manual contains the information you need to get started with the Advantech 32-bit DLL Drivers software package. The DLL Drivers allow you to easily perform versatile I/O operations through properties, methods and events in programs developed with Microsoft Visual Basic, Microsoft Visual C++, Delphi, Borland C++ Builder and other programming languages and development environments.

This manual contains step-by-step instructions for building applications with the DLL Drivers. You can modify these sample applications to suit your needs. This manual does not show you how to solve every possible programming problem. Specific questions should be directed to Advantech's application engineers.

To use this manual, you should already be familiar with one of the supported programming environments and Windows 95 or Windows NT.

Organization of This Manual

This user manual is divided into the following sections:

- Chapter 1, ***Introduction to the 32-bit Windows 95/98/NT DLL Drivers***, introduces the DLL Drivers and how they can be used in your applications to get the most out of Advantech's Data Acquisition and Control cards. It also explains how to install the software. Two additional utilities, the Device Installation Utility (DEVINST.EXE) and the Advantech Test Utility are introduced to set up and test your Advantech hardware. Using DEVINST.EXE to configure your hardware must be completed before you can write programs using the DLL Drivers to access your hardware.
- Chapter 2, ***Creating Windows 95/98 and Windows NT Applications with the DLL Drivers*** briefly explains how to use the DLL Drivers with four popular development environments. It also highlights some programming issues such as buffer allocation, string passing and parameter passing that you should consider during development.

- Chapter 3, ***Tutorial*** gives the new user a walk-through in creating a simple application. Step-by-step procedures are given for a Win32 console application and using the Microsoft Visual Basic and Borland Delphi development environments. In addition, there is a listing of all the sample programs and code that are available on the CD-ROM disc.
- Chapter 4, ***Function Overview*** introduces the kinds of hardware functions that can be programmed by using the DLL Drivers. The functions include *device*, *analog input*, *analog output*, *digital input/output*, *counter*, *temperature measurement*, *alarm*, *port*, *communication* and *event*.
- Chapter 5, ***Functions Reference*** is a listing of all the functions and data structures that are supported by the 32-bit DLL Drivers. In addition, it shows what functions are supported by each of Advantech's hardware models.
- Appendix A, ***Error Codes*** explains the error codes that might be returned when calling functions provided by the DLL Drivers. Refer to this section when debugging your application.

Contents

Figures

Tables

CHAPTER 1

Introduction to the 32-bit Windows 95/98/NT DLL Drivers

1.1 About the Advantech DLL Driver Software

The Advantech DLL driver software provides complete hardware functions and maximum performance. It is freely bundled with all Advantech plug-in I/O cards. With the Advantech DLL driver, you don’t have to use hardware-specific register commands and it gives you a powerful programming API for use with a variety of programming environments and languages.

Advantech DLL driver software supports the high-speed functions that utilize DMA or interrupt for data acquisition. This kind of data transfer is performed in the background. It thus uses less CPU time and speeds the data transfer rate. These functions are used to transfer large amounts of data at high rates. The driver uses double-buffering techniques for continuous, uninterrupted transfer of large amounts of data.

The Advantech DLL driver also supports event functions. It notifies your program by posting messages when events occur within the device. You only have to take the necessary actions when receiving the messages without checking its status manually. It is more efficient and reduces the program’s complexity.

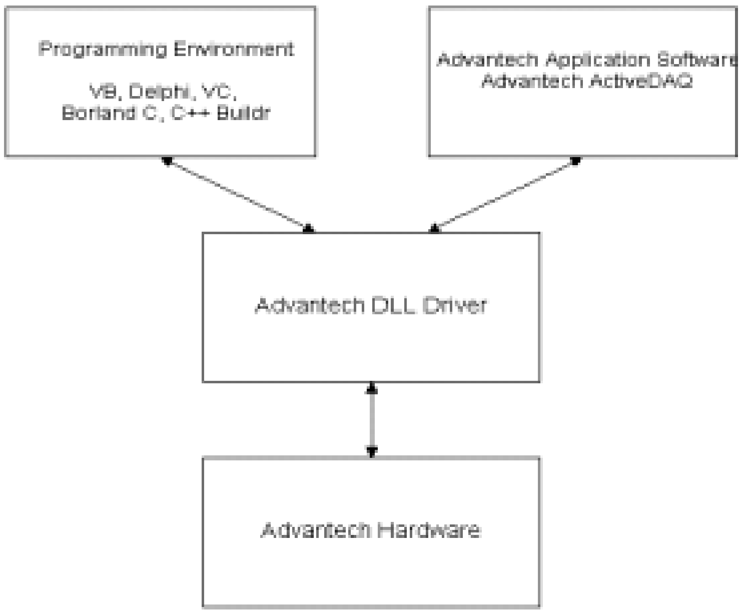


Figure 1-1: DLL Drivers, Programming Environment and Hardware

1.1.1 How To Start Programming Advantech Hardware

The following figure shows the steps to program with Advantech DLL driver software.

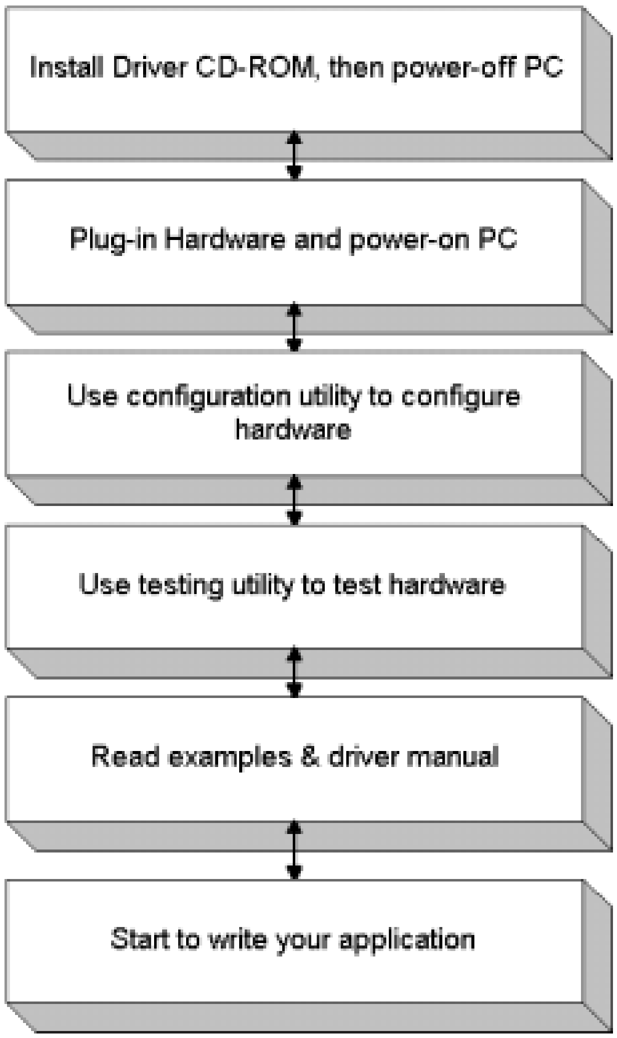

Install Driver CD-ROM, then power-off PC
Plug-in Hardware and power-on PC
Use configuration utility to configure hardware
Use testing utility to test hardware
Read examples & driver manual
Start to write your application

Figure 1-2: Steps to Start Programming with Advantech's DLL Drivers

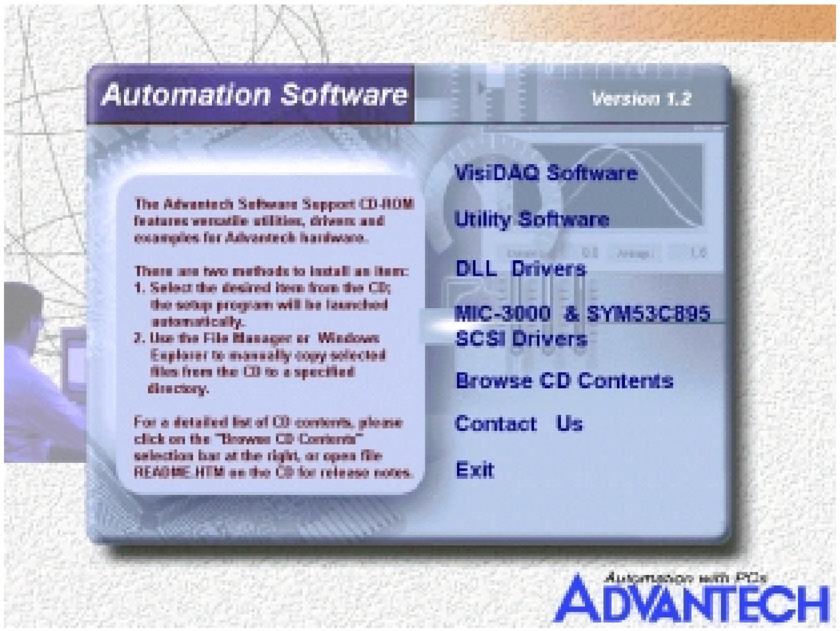
1.2 Installing the 32-bit DLL Drivers

1.2.1 Installing the Software

The installation CD-ROM is shipped with I/O cards. You can use it to install the 32-bit DLL driver. Please follow the steps below to install the driver software:

1. Insert the driver installation CD-ROM disc into your CD-ROM drive.
2. The installation program will run automatically if you have enabled the Windows auto-run feature. If auto-run is not enabled on your computer, use your Windows Explorer or the Windows Run command to execute setup.exe on the driver installation CD-ROM disc (assume “d” is the letter of your CD-ROM disc drive):

```
d:\setup.exe
```



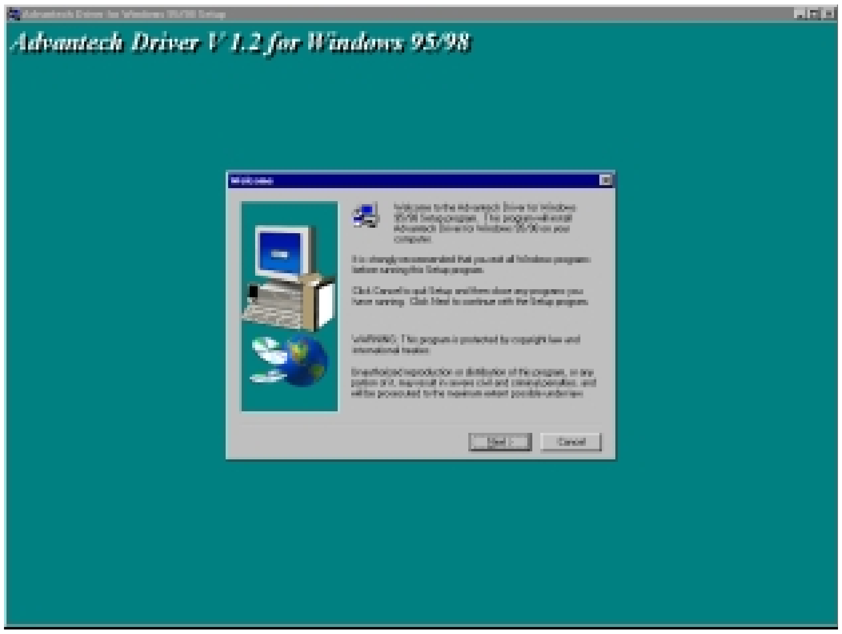
The Advantech Software Support CD-ROM features versatile utilities, drivers and examples for Advantech hardware.

There are two methods to install an item:
1. Select the desired item from the CD; the setup program will be launched automatically.
2. Use the File Manager or Windows Explorer to manually copy selected files from the CD to a specified directory.

For a detailed list of CD contents, please click on the "Browse CD Contents" selection bar at the right, or open file README.HTM on the CD for release notes.

Figure 1-3: Advantech’s Automation Software Installation Program

3. Click the link labeled **DLL Drivers** and then click either the link labeled *Windows 95/98* or *Windows NT* (depending on the platform that is running your development tool).
4. The installation program loads. Click the **Next** button to advance to the following screen.


Advantech Driver for Windows 95/98 Setup
Advantech Driver V 1.2 for Windows 95/98
Welcome
Next >
Cancel

Welcome to the Advantech Driver for Windows 95/98 Setup program. This program will install Advantech Driver for Windows 95/98 on your computer.

It is strongly recommended that you exit all Windows programs before running this Setup program.

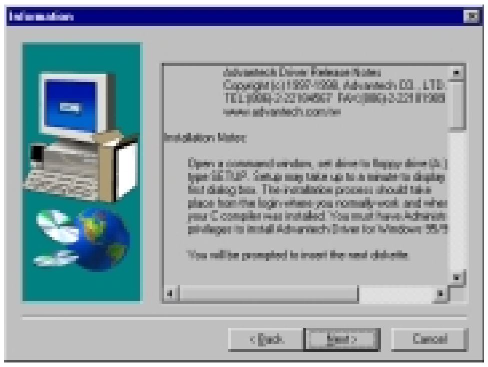
Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

Figure 1-4: DLL Drivers Installation Program Welcome Screen

5. The *Information* screen loads and a window will display the latest release notes. We highly recommend that you read through the release notes since they contain last minute product information that may not appear in the printed manual or online help. When you have finished reading the release notes, click the **Next** button.



Advantech Driver Release Notes
Copyright (c) 1997-1998, Advantech CO., LTD.
TEL:(886)-2-22184567 FAX:(886)-2-22181989
www.advantech.com.tw

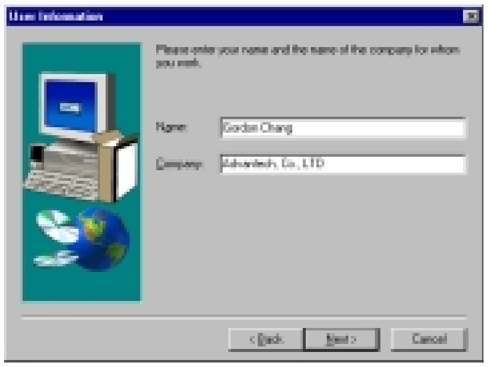
Installation Notes:

Open a command window, set drive to floppy drive (A:)
type SETUP. Setup may take up to a minute to display
first dialog box. The installation process should take
place from the login where you normally work and wher
your C compiler was installed. You must have Administ
privileges to install Advantech Driver for Windows 95/9

You will be prompted to insert the next diskette.

Figure 1-5: Information Screen with Latest Release Notes

6. The *User Information* screen loads. Enter your name and organization's name to personalize your copy of the DLL Drivers program and click the **Next** button.



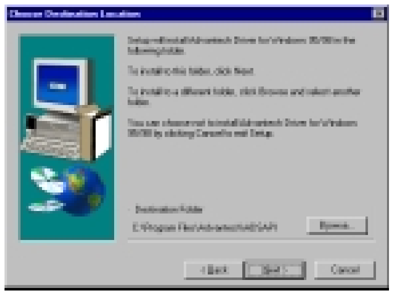
Please enter your name and the name of the company for whom you work.

Name: Gordon Chang

Company: Advantech, Co., LTD

Figure 1-6: Enter Your Name and Company Name

7. You have to select a location on your hard drive where the files will be copied. The default location is c:\Program Files\Advantech\ADSAPI. If you want to save the files to a different path on your hard disk drive, click the **Browse** button and either select or type the path that you want to use. Click the **Next** button to advance.


Choose Destination Location

Setup will install Advantech Driver for Windows 95/98 in the following folder.

To install to this folder, click Next.

To install to a different folder, click Browse and select another folder.

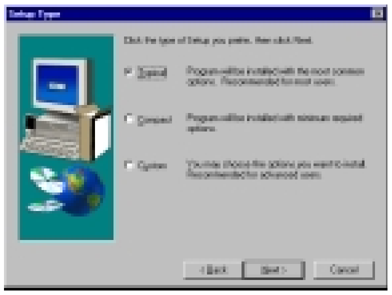
You can choose not to install Advantech Driver for Windows 95/98 by clicking Cancel to exit Setup.

Destination Folder

C:\Program Files\Advantech\ADSAPI

Figure 1-7: Choose Destination Location Dialog Window

8. Choose the kind of installation that you want to perform in the *Setup Type* in the screen. You can select *Typical*, *Compact* or *Custom*. Click the **Next** button to advance.


Setup Type

Click the type of Setup you prefer, then click Next.

[✓ Typical] Program will be installed with the most common options. Recommended for most users.

Compact Program will be installed with minimum required options.

Custom You may choose the options you want to install. Recommended for advanced users.

Figure 1-8: Select the Kind of Installation You Want to Perform

9. The installation program will create a shortcut on your Windows Start menu. The default is *Advantech Driver for 95 and 98* or *Advantech Driver for NT*. If you want to change the name of the shortcut, type your selection in the text box. Click the **Next** button to advance.

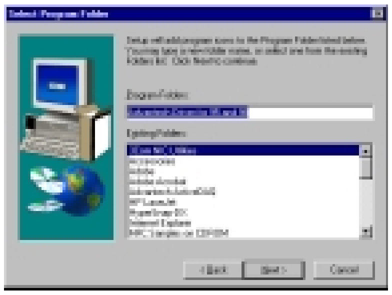


Figure 1-9: Select the Program Folder for the Program Shortcut

10. The installation program will copy the files to your computer.

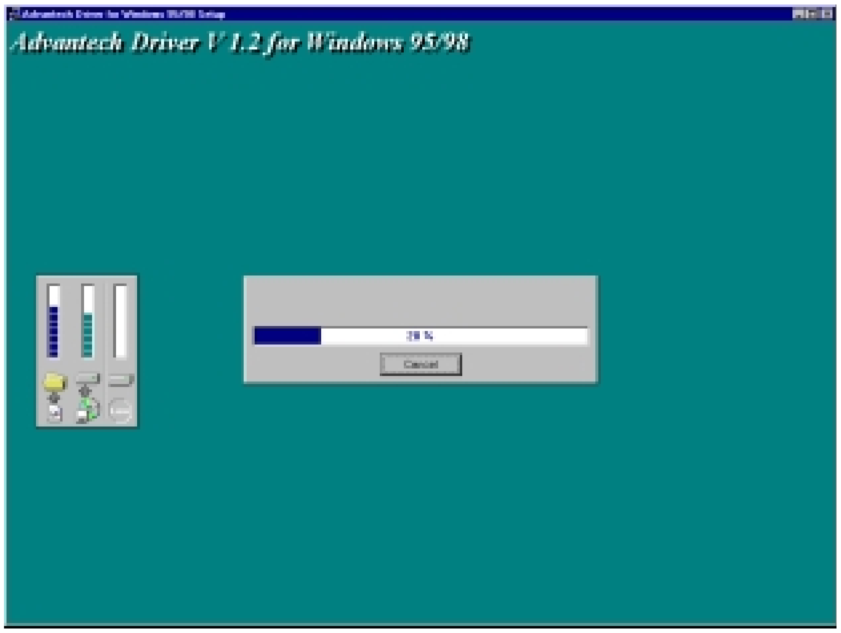


Figure 1-10: Copying the Files

11. The Advantech DLL Drivers are now installed to your computer. Click the **Finish** button to close the installation program.

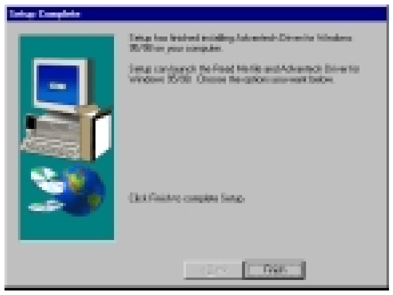


Figure 1-11: Installation Complete

12. At the end of the installation you will be prompted to install the Adobe Acrobat Reader. This software is required to view the online version of the manual. Click the **Yes** button in the message box if you do not yet have this software.

Note: *You can always get the latest version of the Adobe Acrobat Reader by going to Abobe's web site at www.adobe.com.*

13. You can see the shortcuts that were added to your Windows Start menu by choosing **Programs | Advantech Driver for 95 and 98** or **Advantech Driver for NT**.

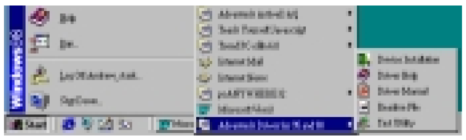


Figure 1-12: Advantech DLL Driver Start Menu Program Shortcuts

1.2.2 File Structure

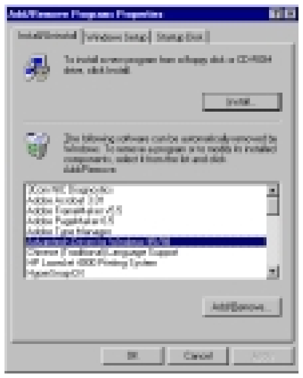
The Advantech DLL Drivers files are installed by default into the c:\Program Files\Advantech\ADSAPI. This root directory contains the Device Installation Utility (DEVINST.EXE), the Advantech Device Testing Utility and many help files for using the DLL Drivers with specific Advantech hardware products.

The subdirectories of this folder contain many examples of using the DLL Drivers with Microsoft Visual Basic, Microsoft Visual C++, Borland Delphi, and other related documentation that will help you get the most of your development projects for Advantech hardware.

1.2.3 Uninstalling the DLL Drivers

The Advantech DLL Drivers include an uninstallation utility to help your remove the software from your computer. To uninstall the software, complete the following procedure:

1. Select **Settings | Control Panel | Add/Remove Programs** from the Windows Start menu. Click the *Install/Uninstall* tab.
2. Highlight the item *Advantech Driver for Windows 95/98* or *Advantech Driver for NT* and then click the **Add/Remove…** button.



To install a new program from a floppy disk or CD-ROM drive, click Install.

The following software can be automatically removed by Windows. To remove a program or to modify its installed components, select it from the list and click Add/Remove.

3Com NIC Diagnostics
Adobe Acrobat 3.01
Adobe FrameMaker v5.5
Adobe PageMaker 6.5
Adobe Type Manager
Advantech Driver for Windows 95/98
Chinese (Traditional) Language Support
HP LaserJet 4000 Printing System
HyperSnap-DX

Figure 1-13: Windows Control Panel Add/Remove Programs Dialog Box

3. When the *Confirm File Deletion?* message box loads, click the **Yes** button.
4. The files will be removed from your computer. When the uninstallation is complete, click the **OK** button in the *Remove Programs From Your Computer* dialog window.

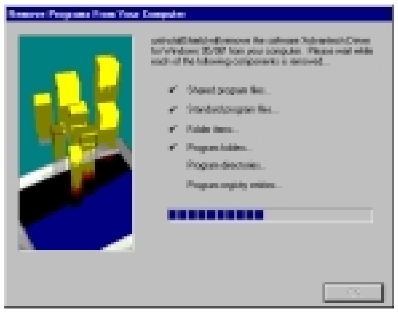


Figure 1-14: Click OK When the Uninstallation Completes

1.3 Device Installation Utility (Configuration Program)

The configuration utility is a software program that allows you to configure your hardware and store the settings in your Windows registry. These settings will be used when you call the APIs of the Advantech 32-bit DLL drivers.

To set up or configure each I/O device within the configuration utility, complete the following procedure.

1. Select **Setup | Device** from the main menu.

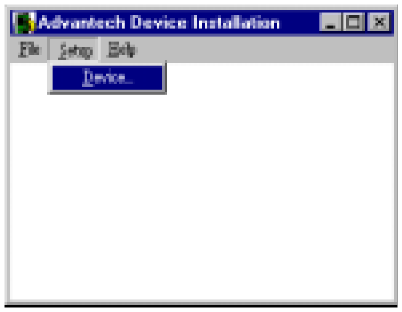


Figure 1-15: Installing a Device in the Device Installation Utility

2. You can see the installed devices in the *I/O Device Installation* window.

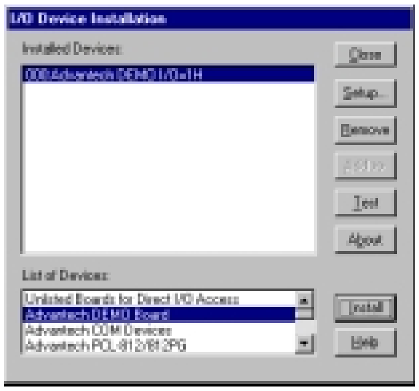


Figure 1-16: I/O Device Installation Window

3. Scroll down the *List of Devices* list box and select the device that you want to install. Click the **Install** button

Note *If there are any existing devices of that type installed on your computer, a dialog box will load to display them. Select the device that you want to operate from the listing and press the* ***OK*** *button.*

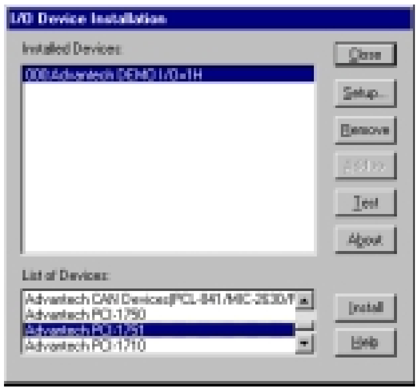
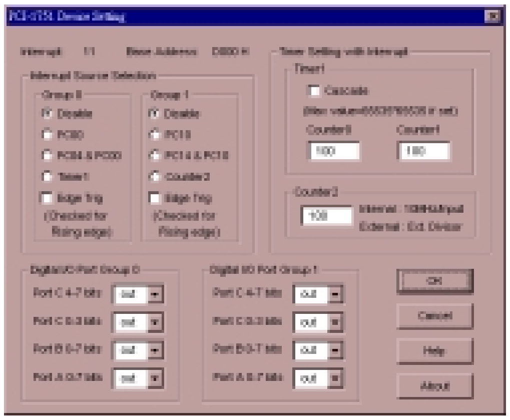


Figure 1-17: Existing Device Found

4. A configuration dialog box will load for the specified device.


PCI-1751 Device Setting

Interrupt: 11 Base Address: D800 H

Interrupt Source Selection

Group 0: (● Disable) PC00, PC04 & PC00, Timer1; Edge Trig (Checked for Rising edge)

Group 1: (● Disable) PC10, PC14 & PC10, Counter2; Edge Trig (Checked for Rising edge)

Timer Setting with Interrupt

Timer1: Cascade (Max value=65535*65535 if set)

Counter0: 100 Counter1: 100

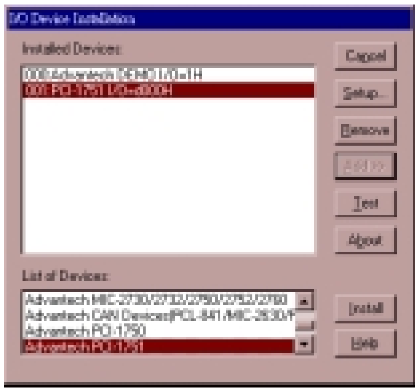
Counter2: 100 Internal : 10MHz input External : Ext. Device

Digital I/O Port Group 0	
Port C 4-7 bits	out
Port C 0-3 bits	out
Port B 0-7 bits	out
Port A 0-7 bits	out

Digital I/O Port Group 1	
Port C 4-7 bits	out
Port C 0-3 bits	out
Port B 0-7 bits	out
Port A 0-7 bits	out

Figure 1-18: Installed Device Configuration Window

5. After you configure the device, click the **OK** button and the device will be shown in the *Installed Devices* field as below.


I/O Device Installation

Installed Devices:

000:Advantech DEMO I/O=1H
001:PCI-1751 I/O=d000H

List of Devices:

Advantech MIC-2730/2732/2750/2752/2760
Advantech CAN Devices(PCL-841/MIC-2630/P
Advantech PCI-1750
Advantech PCI-1751

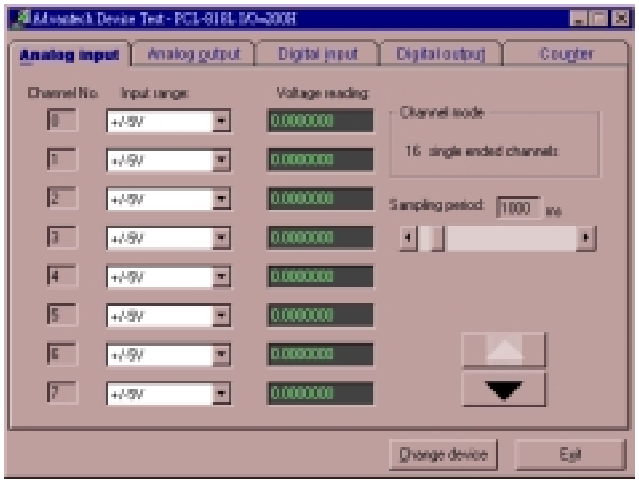
Figure 1-19: Newly Installed Device

Note: *The device number of the installed device is 001 that is the prefix of “001:PCI-1751 I/O=d000H”. You must pass the device number to the driver to specify the device that you wish to operate.*

1.4 Device Test Utility

After configuring your hardware with the Device Installation Utility, you can use the testing utility to verify the hardware functions. The test utility provides analog input, analog output, digital input, digital output, and counter functions. You can launch it in the Device Installation Utility by highlighting the device that you want to test in the *I/O Device Installation* window and then clicking the **Test** button. Alternatively, you can load it by clicking on the *Test Utility* shortcut in the *Advantech DLL* driver folder on your Windows Start menu.

Click on the *Analog input* tab in the test utility and you can see the analog input test panel as below. Select the input range for each channel in the *Input range* field. Configure the sampling rate in the scroll bar. Switch the channels with the up or down arrow.



Channel No.	Input range:	Voltage reading:
0	+/-5V	0.0000000
1	+/-5V	0.0000000
2	+/-5V	0.0000000
3	+/-5V	0.0000000
4	+/-5V	0.0000000
5	+/-5V	0.0000000
6	+/-5V	0.0000000
7	+/-5V	0.0000000

Channel mode: 16 single ended channels

Sampling period: 1000 ms

Figure 1-20: Advantech Device Test Utility (Analog input tab)

Click on the *Analog output* tab to switch to the analog output test panel as shown below. It allows you to output sine, square, and triangle waveforms automatically, or output a single value manually. You can configure the waveform frequency and switch the output channel by using the up or down button.

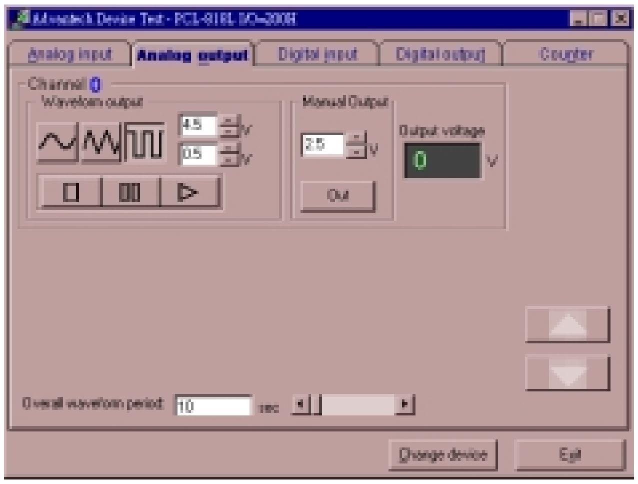


Figure 1-21: Advantech Device Test Utility (Analog output tab)

Click on the *Digital input* tab to switch to the digital input test panel as shown below. You can use the up or down arrows to switch the digital input channel.

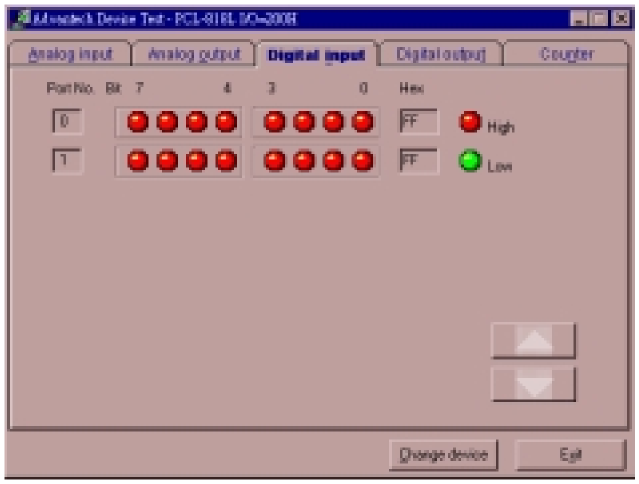


Figure 1-22: Advantech Device Test Utility (Digital input tab)

Click on the *Digital output* tab to switch to the digital output test panel as shown below. By pressing the buttons on the panel, you can output the desired values to the corresponding ports. You can use the up or down arrows to switch the digital output channel.

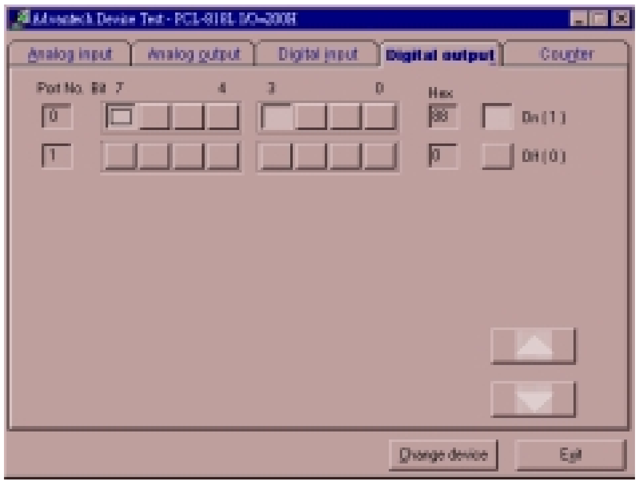
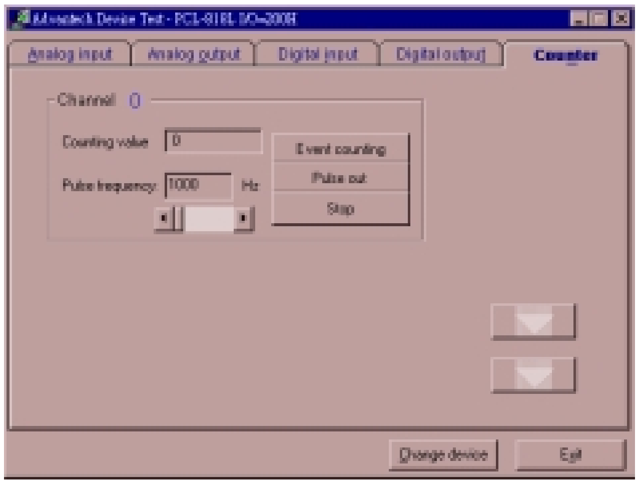


Figure 1-23: Advantech Device Test Utility (Digital output tab)

Click on the *Counter* tab to switch to the counter test panel as shown below. It provides event counting and pulse output functions. You can configure the pulse frequency by using the scroll bars. You can also switch the counter channel by using the up or down arrows.



Channel 0

Counting value: 0

Pulse frequency: 1000 Hz

Figure 1-24: Advantech Device Test Utility (Counter tab)

CHAPTER 2

Creating Windows 95/98 and Windows NT Applications with the DLL Drivers

2.1 Introduction to Programming with the DLL Driver

This section contains general information about building DA&C applications. It describes the nature of the DLL drivers files used in building DA&C applications and explains the basics of making DA&C applications using the following tools:

- Microsoft Visual C++ for Windows 95/98/NT version 5.0
- Microsoft Visual Basic for Windows 95/98/NT version 5.0
- Borland Delphi for Windows 95/98/NT version 4.0
- Borland C++ 5.0 or C++ Builder version 1.0

If you are not using the tools listed, consult your development tool reference documentation for details on creating applications that call DLLs.

The 32-bit Windows 95/98/NT DLL function libraries are dynamically linked, which means that DLL routines are not linked into the executable files of applications. Information about the DLL routines in the DLL import libraries is stored in the executable files. The DLL driver is linked to the application only when DLL functions are called during execution.

Import libraries (*.LIB) contain information about their DLL export functions. They indicate the presence and location of the DLL routines. Depending on the development tool that you are using, you may give the DLL routines information through import libraries or through function declarations.

Using functional prototypes is good programming practice. That is why DLL drivers are packaged with functional prototype files for different Windows development tools. The installation utility copies the appropriate prototype files for the development tools that you choose. If you are not using any of the development tools that Advantech DLL drivers support, you must create your own functional prototype file.

2.2 Using the Win32 Console

This section assumes that you will be using the Microsoft Visual Workbench to manage your code development. Advantech DLL drivers support Microsoft Visual C++ version 2.0 and above.

To use the DA&C functions, you must use the DLL routines. Follow this procedure:

1. Create your source files as you would for other Windows programs written in C++ by calling DLL functions as typical function calls.
2. Prototype any DLL routines used in your application. Include the DLL header file, which prototypes all DLL routines, as shown in the following example:

```
#include “driver.h”
```

3. Add the DLL import library (for example, “*ADSAPI32.LIB*”) to the project module.

For a general outline of creating a Visual C++ Windows programs, complete the following procedure:

1. Click **File | New** from the main menu to create your application project and source code as you would for any other Visual C++ program.

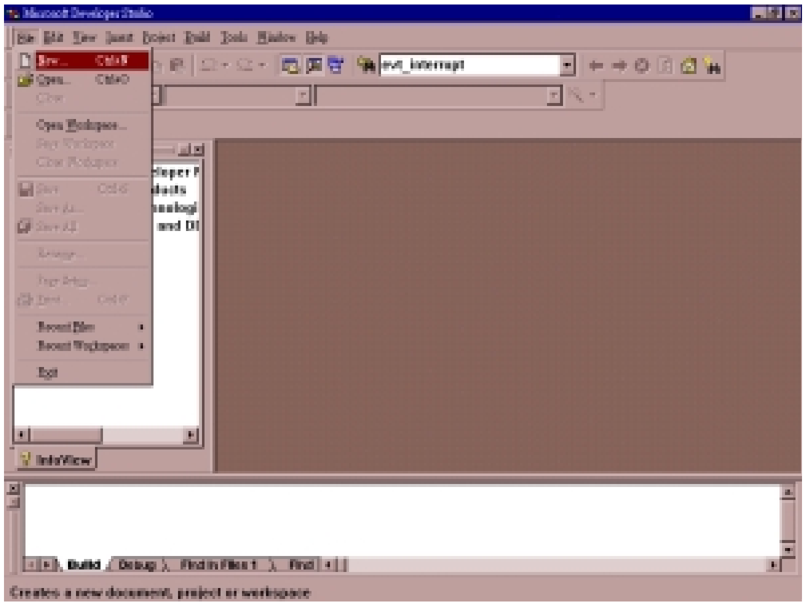


Figure 2-1: Creating a New Visual C++ Application

2. Define the type of new project as "*Win32 Application*", define the platform to be "*Win32*" and assign a project file directory.

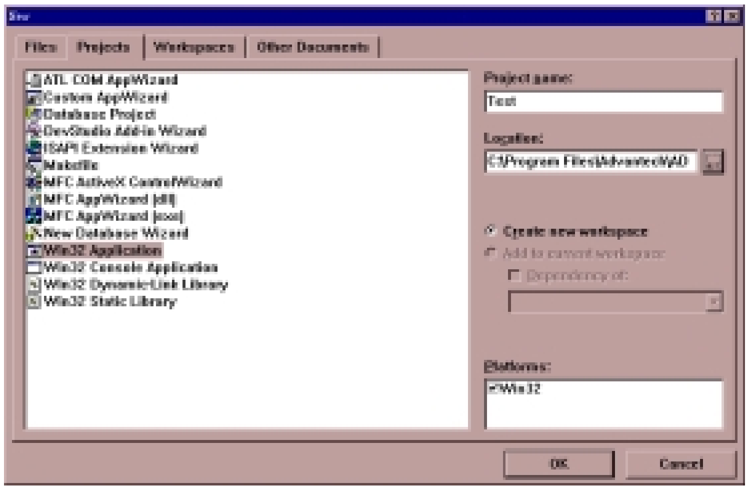


Figure 2-2: Defining the Application Type and Assigning a Project Directory

3. In order to develop DA&C applications with Advantech DLL drivers, you have to first include the Advantech DLL driver for Visual C++ header files (filename.H). The way to include the header file into your project is to select **Project | Add to Project | Files...** from the Visual C++ main menu.

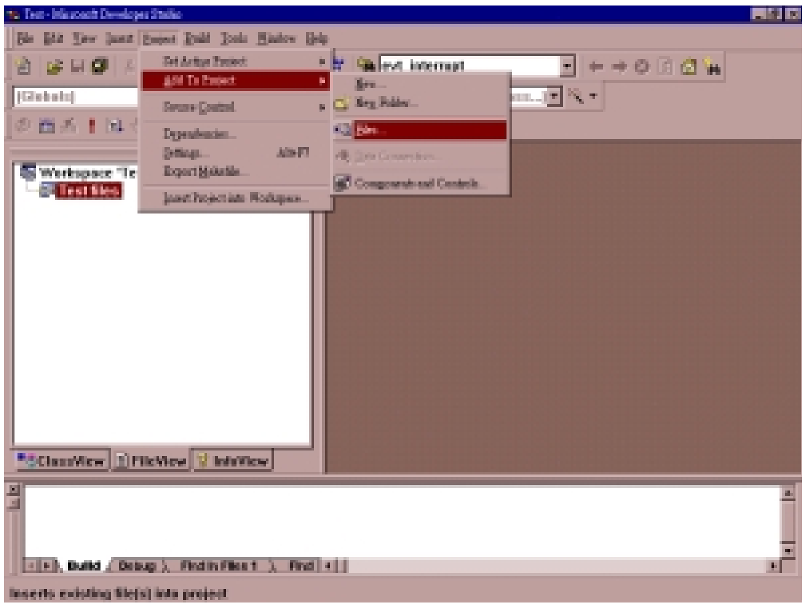


Figure 2-3: Including the DLL Drivers Header File in Your Project

4. After adding the header file for DLL functions, you will see a filename, i.e., "**Driver.h**" listed under your project directory.

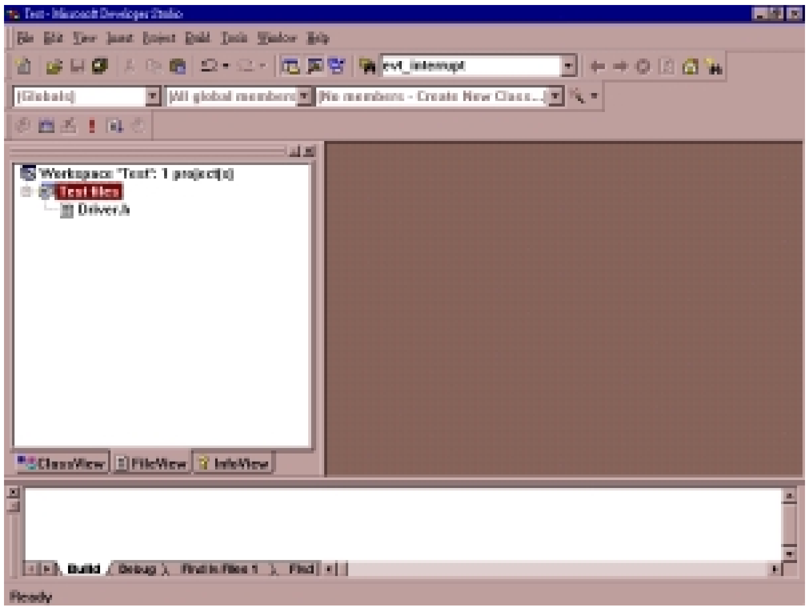
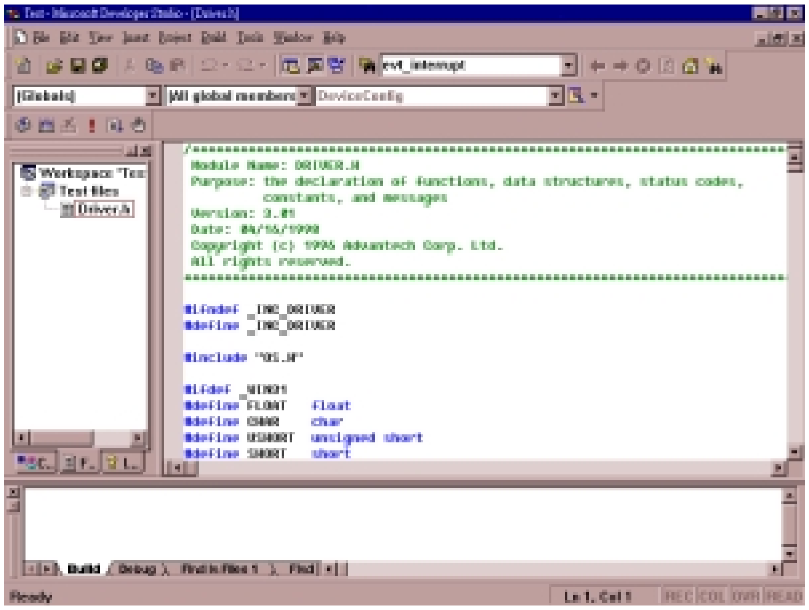

Test - Microsoft Developer Studio
File Edit View Insert Project Build Tools Window Help
evt_interrupt
(Globals)
(All global members)
(No members - Create New Class...)
Workspace 'Test': 1 project(s)
Test files
Driver.h
ClassView
FileView
InfoView
Build
Debug
Find in Files 1
Find
Ready

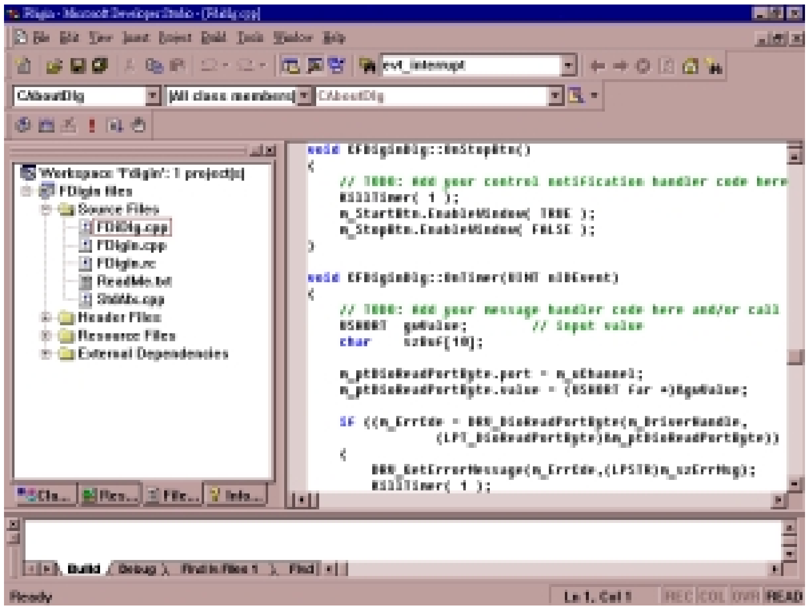
Figure 2-4: Including Driver.h DLL Driver Header File in the Project Directory



```
/*************************************************************************
 Module Name: DRIVER.H
 Purpose: the declaration of functions, data structures, status codes,
          constants, and messages
 Version: 3.01
 Date: 04/16/1998
 Copyright (c) 1996 Advantech Corp. Ltd.
 All rights reserved.
*************************************************************************

#ifndef _INC_DRIVER
#define _INC_DRIVER

#include "OS.H"

#ifdef _WIN31
#define FLOAT    float
#define CHAR     char
#define USHORT   unsigned short
#define SHORT    short
```

Figure 2-5: Driver.h DLL Driver Header File in the Project Directory

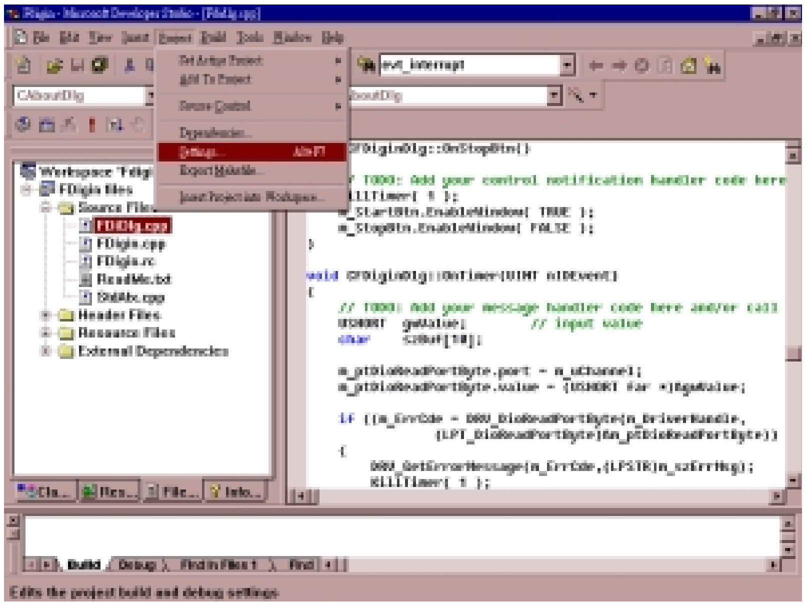
5. After adding the header file, you can view the DA&C constant definition, parameter declaration and DLL function calls that are defined in this header file. These definitions can all be used in your application programs.
6. Create your application source code by clicking **Project | Add to Project | New** and selecting the *C++ Source File* option. After you have created the application, you can also add the related resources and save them into a .rc file and add the .rc file to the project. For more detailed program development information, please refer to the Microsoft Visual C++ User’s Manual.



```
void CFDiginDlg::OnStopBtn()
{
    // TODO: Add your control notification handler code here
    KillTimer( 1 );
    m_StartBtn.EnableWindow( TRUE );
    m_StopBtn.EnableWindow( FALSE );
}

void CFDiginDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call
    USHORT  gwValue;        // input value
    char    szBuf[10];

    m_ptDioReadPortByte.port = m_uChannel;
    m_ptDioReadPortByte.value = (USHORT far *)&gwValue;

    if ((m_ErrCde = DRV_DioReadPortByte(m_DriverHandle,
                (LPT_DioReadPortByte)&m_ptDioReadPortByte))
    {
        DRV_GetErrorMessage(m_ErrCde,(LPSTR)m_szErrMsg);
        KillTimer( 1 );
```

Figure 2-6: Creating Your Application Source Code in the App Studio

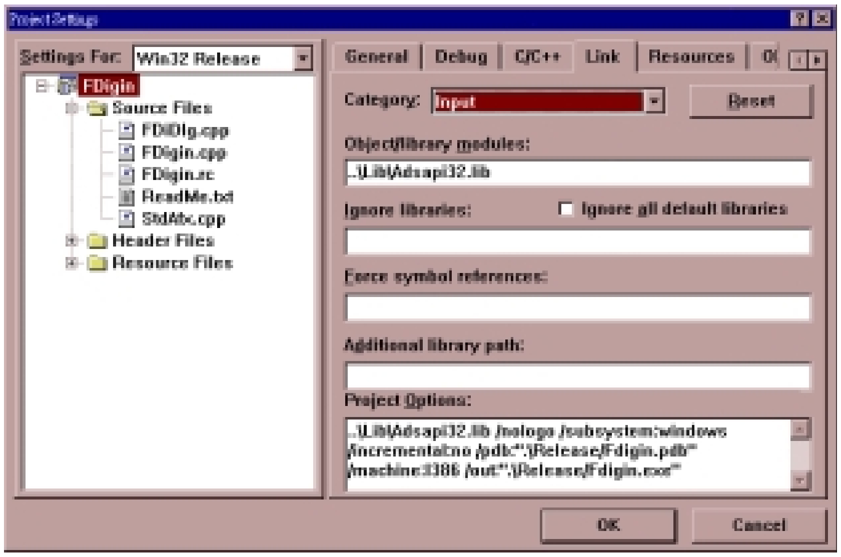
7. Add the DLL import library (for example, "*ADSAPI32.LIB*") into this project by clicking on the ***Build*** menu and choosing the ***Settings*** option. The *Project Setting* dialog box will be displayed at the front of the Windows screen.



```
CFDiginDlg::OnStopBtn()

// TODO: Add your control notification handler code here
KillTimer( 1 );
m_StartBtn.EnableWindow( TRUE );
m_StopBtn.EnableWindow( FALSE );
}

void CFDiginDlg::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call
    USHORT  gwValue;        // input value
    char    szBuf[10];

    m_ptDioReadPortByte.port = m_uChannel;
    m_ptDioReadPortByte.value = (USHORT far *)&gwValue;

    if ((m_ErrCde = DRV_DioReadPortByte(m_DriverHandle,
                (LPT_DioReadPortByte)&m_ptDioReadPortByte))
    {
        DRV_GetErrorMessage(m_ErrCde,(LPSTR)m_szErrMsg);
        KillTimer( 1 );
```

Figure 2-7: Adding the DLL Import Library Into Your Project

8. Select the *Link* page in the *Project Settings* pop-up dialog box and then set the *Category* field to be *Input*. Insert **adsapi32.lib** in *Object/library* modules and click the **OK** button. The DLL driver library will be linked with your application object file and be built into an application execution file through the ***Build*** menu.



Category: Input

Object/library modules:
..\Lib\Adsapi32.lib

Ignore libraries:

☐ Ignore all default libraries

Force symbol references:

Additional library path:

Project Options:

```
..\Lib\Adsapi32.lib /nologo /subsystem:windows
/incremental:no /pdb:".\Release/Fdigin.pdb"
/machine:I386 /out:".\Release/Fdigin.exe"
```

Figure 2-8: Linking Your DLL Driver Libray to Your Project

2.3 Using Microsoft Visual Basic

To use the DA&C functions, you must use the DA&C DLL. Follow this procedure:

1. Select **File | New Project** from the main menu to create your forms and code as you would for any other Visual Basic program.

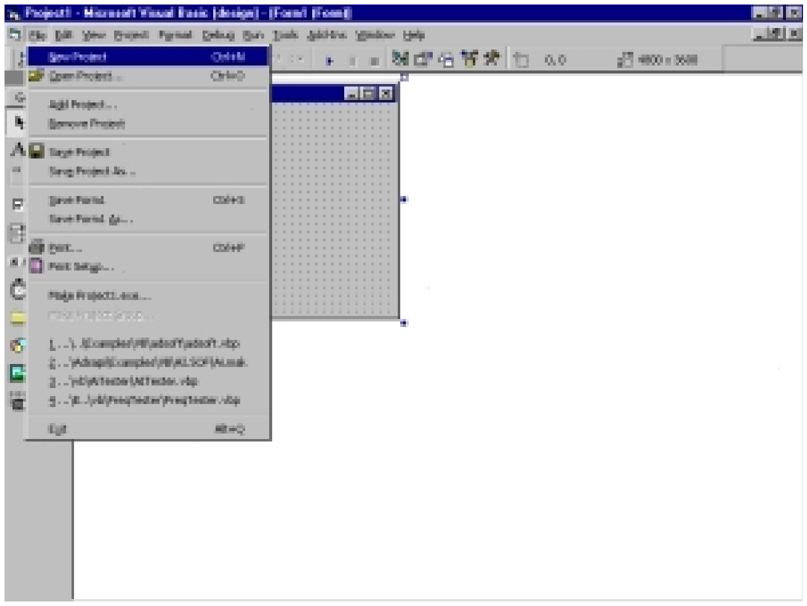


Figure 2-9: Creating a New Visual Basic Project

2. In order to develop a DA&C application with Advantech DLL drivers, you have to first include the Advantech DLL driver for Visual Basic header file. The way to include the header file into your project is to select **View | Project Explorer**.

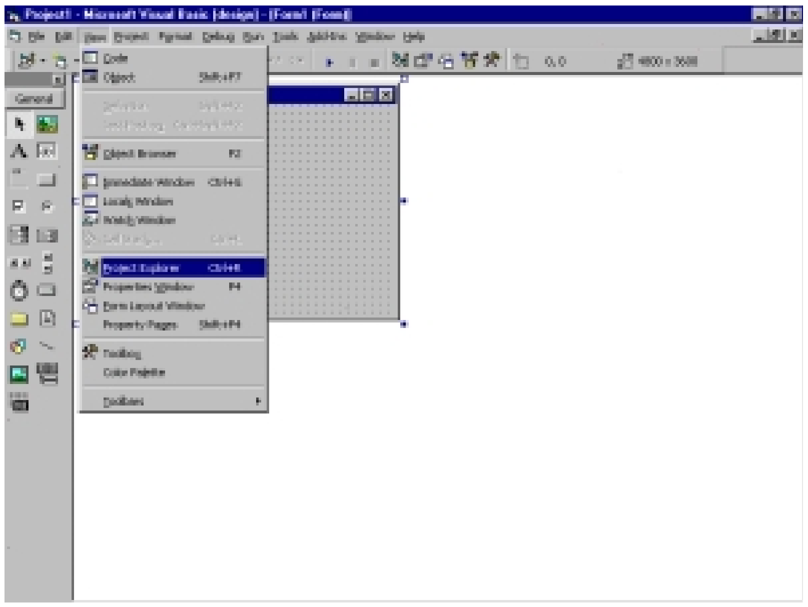


Figure 2-10: Including the Advantech DLL Driver for Visual Basic Header File

3. After clicking on the **Project Explorer**, you will get a window titled with the project name. Move your cursor into this window and right-click to display a list of the available options.

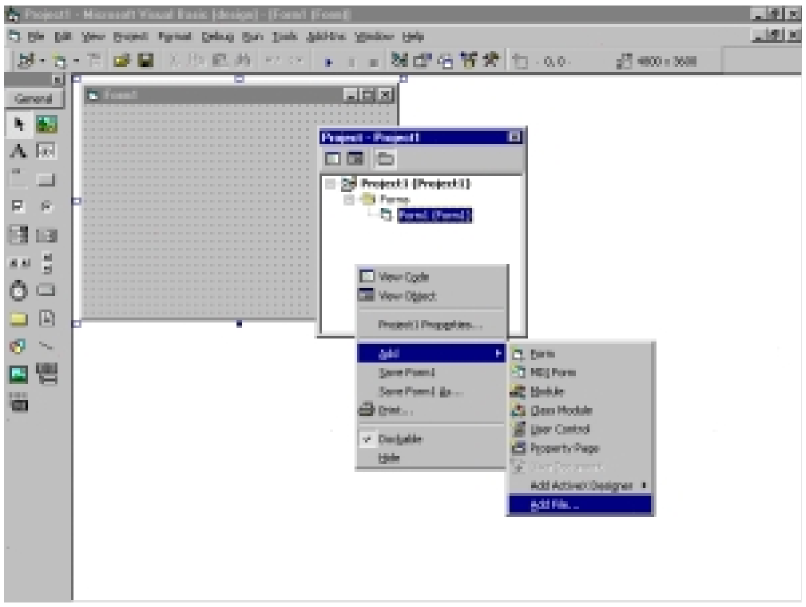


Figure 2-11: Visual Basic Project Options

4. Click on ***Add File*** to include the Visual Basic header file "***Driver.bas***" for Windows 95/98/NT DLL functions. Visual Basic will display a file search window to look for the "Driver.bas" file. This header file is added into the dedicated path assigned by the DLL driver installation. For the Visual Basic DLL calling method, please refer to the Microsoft Visual Basic user manual.

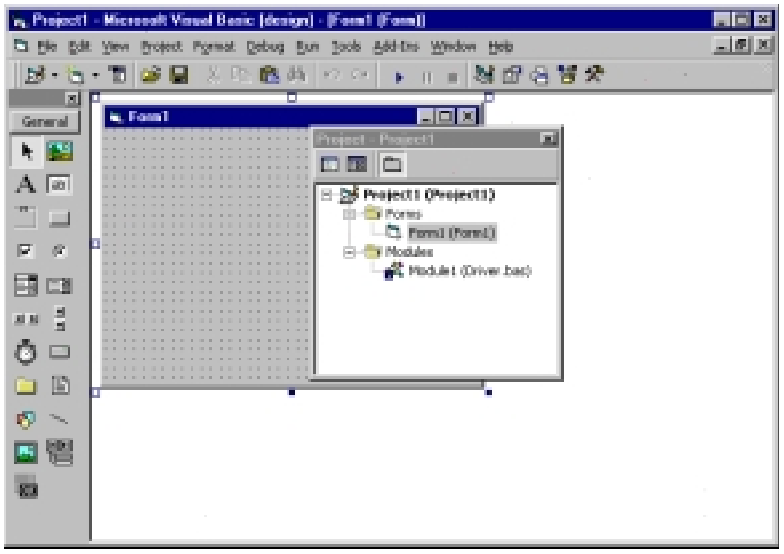
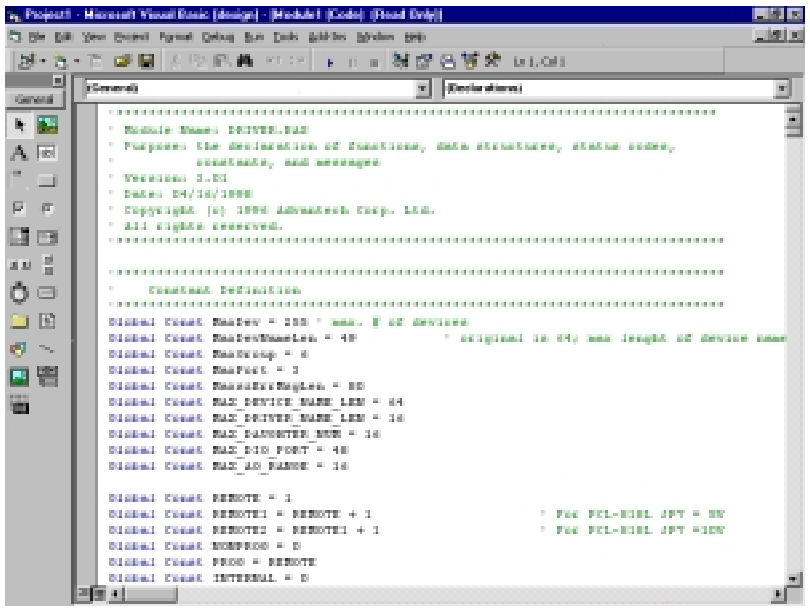

Project1 - Microsoft Visual Basic [design] - [Form1 (Form)]
File Edit View Project Format Debug Run Tools Add-Ins Window Help
General
Form1
Project - Project1
Project1 (Project1)
Forms
Form1 (Form1)
Modules
Module1 (Driver.bas)

Figure 2-12: Adding the Driver.bas Visual Basic Header File to Your Project

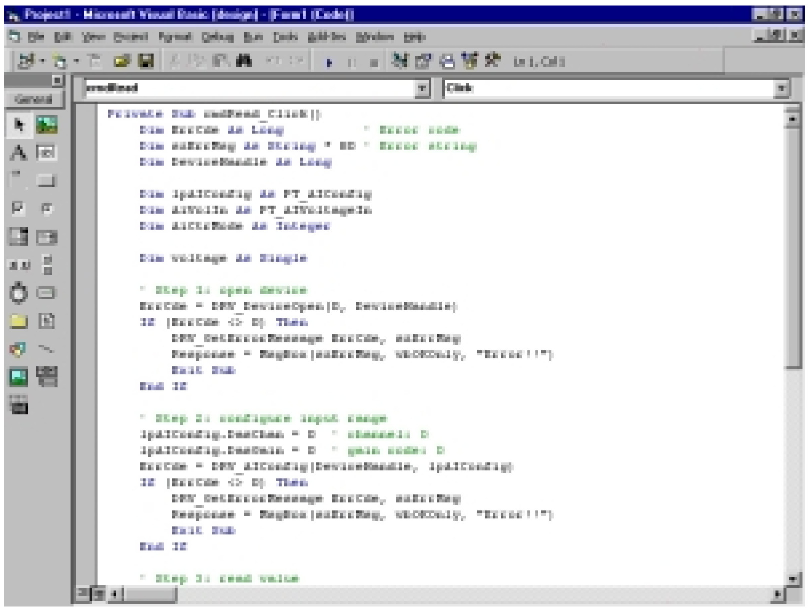
5. After adding the header file, you can view the constant definition, parameter declaration and DLL function calls that you used in your application.


Project1 - Microsoft Visual Basic [design] - [Module1 (Code) (Read Only)]
File Edit View Project Format Debug Run Tools Add-Ins Window Help
Ln 1, Col 1
General
(General)
(Declarations)

```
'****************************************************************************
' Module Name: DRIVER.BAS
' Purpose: the declaration of functions, data structures, status codes,
'          constants, and messages
' Version: 3.01
' Date: 04/16/1998
' Copyright (c) 1996 Advantech Corp. Ltd.
' All rights reserved.
'****************************************************************************

'****************************************************************************
'     Constant Definition
'****************************************************************************
Global Const MaxDev = 255 ' max. # of devices
Global Const MaxDevNameLen = 49          ' original is 64; max lenght of device name
Global Const MaxGroup = 6
Global Const MaxPort = 3
Global Const MaxErrMsgLen = 80
Global Const MAX_DEVICE_NAME_LEN = 64
Global Const MAX_DRIVER_NAME_LEN = 16
Global Const MAX_DAUGHTER_NUM = 16
Global Const MAX_DIO_PORT = 48
Global Const MAX_AO_RANGE = 16

Global Const REMOTE = 1
Global Const REMOTE1 = REMOTE + 1                    ' For PCL-818L JP7 = 5V
Global Const REMOTE2 = REMOTE1 + 1                   ' For PCL-818L JP7 =10V
Global Const NONPROG = 0
Global Const PROG = REMOTE
Global Const INTERNAL = 0
```

Figure 2-13: Constants, Parameters and DLL Function Calls

6. Create your application source code and add DA&C function calls into your application source code as below:


Project1 - Microsoft Visual Basic [design] - [Form1 (Code)]
File Edit View Project Format Debug Run Tools Add-Ins Window Help
Ln 1, Col 1
General
cmdRead
Click

```
Private Sub cmdRead_Click()
    Dim ErrCde As Long          ' Error code
    Dim szErrMsg As String * 80 ' Error string
    Dim DeviceHandle As Long

    Dim lpAIConfig As PT_AIConfig
    Dim AIVolIn As PT_AIVoltageIn
    Dim AICtrMode As Integer

    Dim voltage As Single

    ' Step 1: open device
    ErrCde = DRV_DeviceOpen(0, DeviceHandle)
    If (ErrCde <> 0) Then
        DRV_GetErrorMessage ErrCde, szErrMsg
        Response = MsgBox(szErrMsg, vbOKOnly, "Error!!")
        Exit Sub
    End If

    ' Step 2: configure input range
    lpAIConfig.DasChan = 0  ' channel: 0
    lpAIConfig.DasGain = 0  ' gain code: 0
    ErrCde = DRV_AIConfig(DeviceHandle, lpAIConfig)
    If (ErrCde <> 0) Then
        DRV_GetErrorMessage ErrCde, szErrMsg
        Response = MsgBox(szErrMsg, vbOKOnly, "Error!!")
        Exit Sub
    End If

    ' Step 3: read value
```

Figure 2-14: Writing the Source Code

The DLL driver for Windows 95/98/NT function library includes a DLL and a Windows 95/98/NT kernel device driver (SYS files). Most of the functions are built into the kernel mode device driver. The DLL provides a portable programming interface for DA&C applications.

The DLL driver for Windows 95/98/NT, ADSAPI32.DLL, is located in your Windows 95/98 *System* directory or your Windows NT *System32* directory. The kernel device driver, ADSIO.SYS, is found in the Windows\Drivers directory.

In addition to the system files, the DLL Driver for Windows 95/98/NT is shipped with other files to help you develop DA&C applications. These files are in the directory you select when installing the DLL Drivers for Windows 95/98/NT.

Note: *"Driver.bas" is an include file that contains all DLL function prototypes. You should include this file in your source files when you build your DA&C applications.*

2.4 Using Borland Delphi

To use the DA&C functions, you must use the DA&C DLL. Follow this procedure:

1. Select **File | New Application** option from the Borland Delphi main menu to create your forms and code as you would for any Borland Delphi application program.

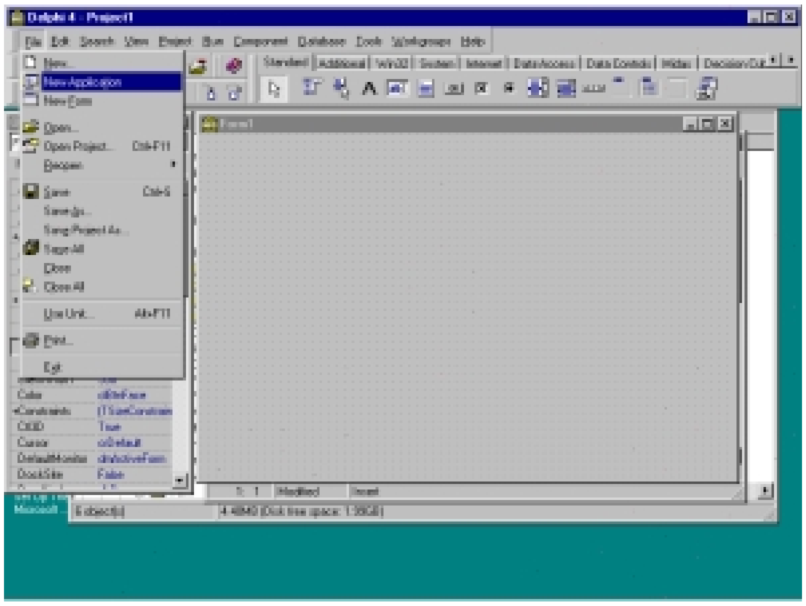


Figure 2-15: Creating a New Borland Delphi Application

2. After creating a new project, choose the ***View*** menu and select the ***Project Manager*** option to design your application forms and write the code. A *Project Manager* pop-up dialog will be shown on the screen.

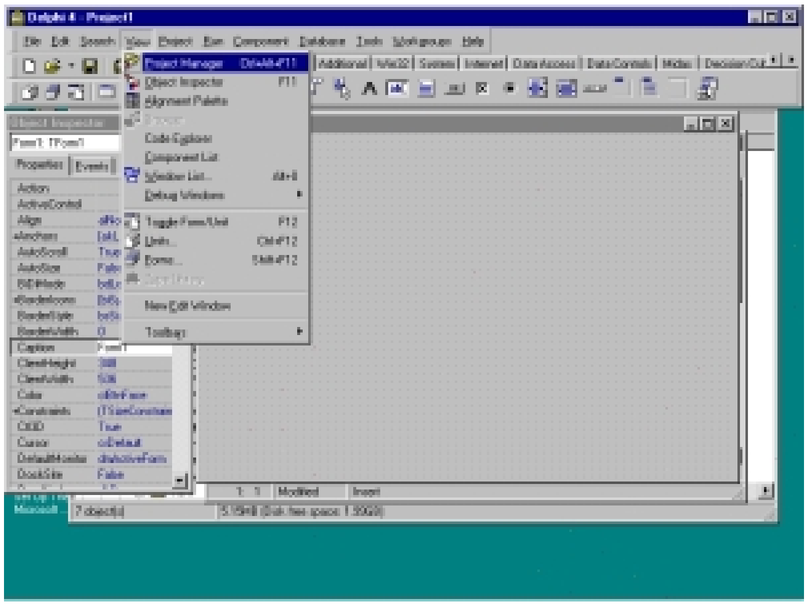
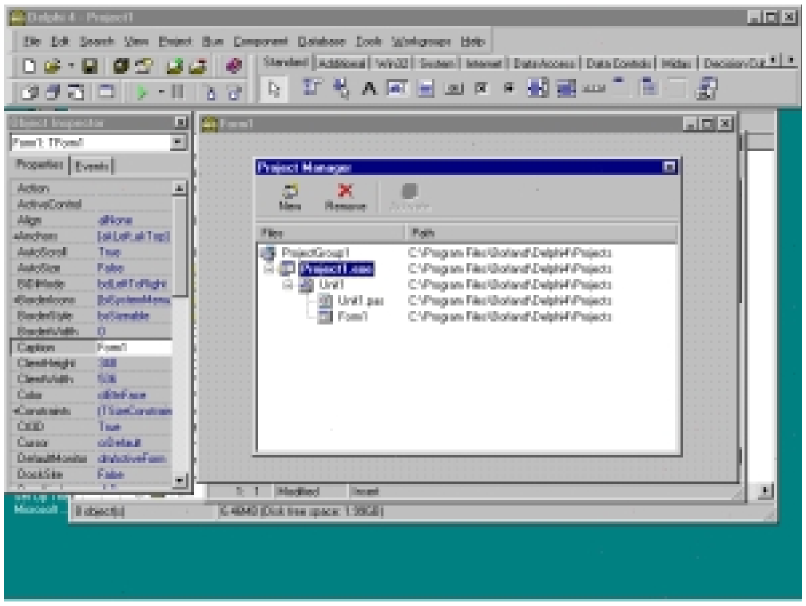


Figure 2-16: Opening the Project Manager

3. You should first add the Advantech 32-bit DLL driver header file into your application project by clicking **Project | Add to Project** from the main menu. Borland Delphi will display a search dialog box to select the location of the DLL header file.

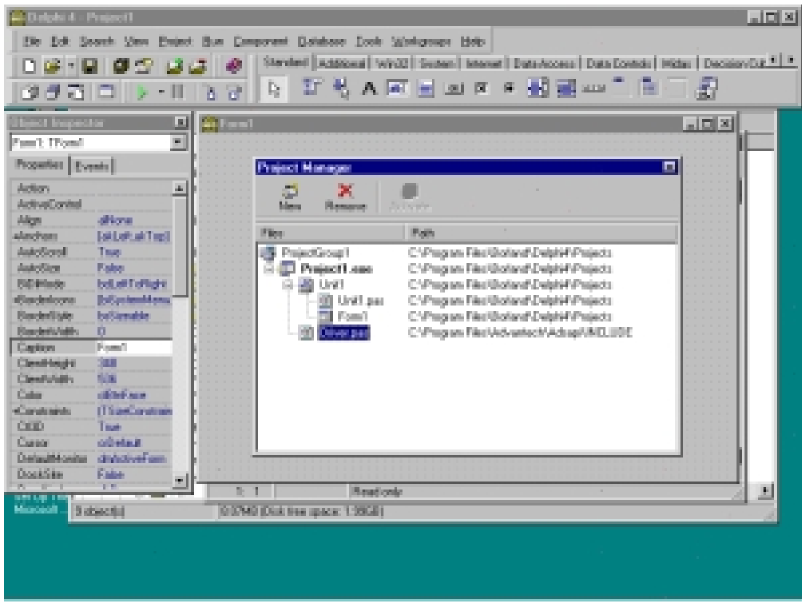


Delphi 4 - Project1
File Edit Search View Project Run Component Database Tools Workgroups Help
Standard Additional Win32 System Internet DataAccess Data Controls Midas Decision Cube
Object Inspector
Form1
Project Manager
New
Remove
Activate

Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi4\Projects
Project1.exe	C:\Program Files\Borland\Delphi4\Projects
Unit1	C:\Program Files\Borland\Delphi4\Projects
Unit1.pas	C:\Program Files\Borland\Delphi4\Projects
Form1	C:\Program Files\Borland\Delphi4\Projects

Figure 2-17: Adding Header File Into Your Project

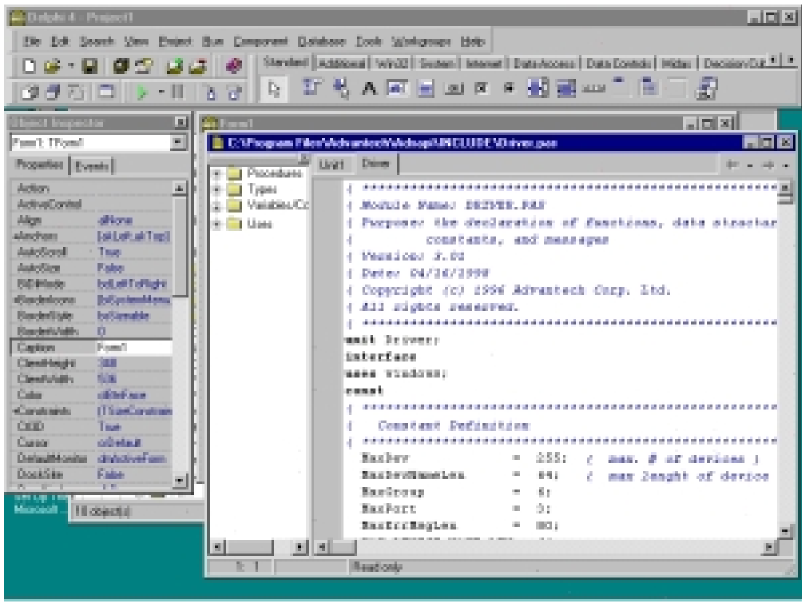
4. Select the DLL header file (“***Driver.pas***”) and click the **OK** button. The *Project Manager* dialog box will show this header file and its location.



Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi4\Projects
Project1.exe	C:\Program Files\Borland\Delphi4\Projects
Unit1	C:\Program Files\Borland\Delphi4\Projects
Unit1.pas	C:\Program Files\Borland\Delphi4\Projects
Form1	C:\Program Files\Borland\Delphi4\Projects
Driver.pas	C:\Program Files\Advantech\Adsapi\INCLUDE

Figure 2-18: Adding the DLL Header File

5. Double-click the header file. You will find the constant definitions, parameter declarations and DLL function calls that can be used in your DA&C applications.



```
{ ****************************************************************
{ Module Name: DRIVER.PAS
{ Purpose: the declaration of functions, data structur
{           constants, and messages
{ Version: 3.02
{ Date: 04/24/1998
{ Copyright (c) 1996 Advantech Corp. Ltd.
{ All rights reserved.
{ ****************************************************************
unit Driver;
interface
uses windows;
const
{ ****************************************************************
{   Constant Definition
{ ****************************************************************
  MaxDev              =  255;   {  max. # of devices }
  MaxDevNameLen       =  64;    {  max lenght of device
  MaxGroup            =  6;
  MaxPort             =  3;
  MaxErrMsgLen        =  80;
```

Figure 2-19: Constants, Parameters and DLL Function Calls

6. Create your forms and Delphi program code. You can find lots of example programs and related source code included on the DLL Drivers CD-ROM disc. After designing the program code, click the **Run** button to test or build the application execution file.

The DLL driver for the Windows 95/98/NT function library includes a DLL driver and a Windows 95/98/NT kernel device driver (SYS files). Most of the functionality is built into the kernel mode device driver. The DLL provides a portable programming interface for DA&C applications.

The DLL driver for Windows 95/98/NT, ADSAPI32.DLL, is located in your Windows 95/98/NT \System or \System32 directories. The kernel device driver, ADSIO.SYS, can be found in your Windows 95/98/NT \Drivers directory.

In addition to the system files, the DLL Driver for Windows 95/98/NT is shipped with other files to help you develop DA&C applications. These files are in the directory you select when installing the DLL Driver for Windows 95/98/NT.

Note: *"Driver.pas" is an include file that contains all DLL function prototypes. You should include this file in your source files when you build your DA&C applications.*

2.5 Using Borland C++ or Borland C++ Builder

To use the DA&C functions, you must use the DA&C DLL. Follow this procedure:

1. Create a new project using the *Class library*. Do not choose OWL if you want to include the sample program code (SDK).
2. If the header library (for example, ***\ADSAPI\EXMPLES\BC\LIB\ADSAPIBC.LIB***) is not compatible with your Borland C++ version, you should apply the **IMPLIB** utility to create a new **.LIB** file.

```
Syntax: IMPLIB [options]   libname[.lib]   dllname[.dll]
```

Example:

```
IMPLIB C:\Program Files\Advantech\Adsapi\Examples\BC\LIB
\ADSAPIBC.LIB
C:\WinNT\ADSAPI32.DLL
```

Note: The header file for Borland C++ and Borland C++ Builder is identical to the Visual C++ header file.

1. Add your source code filename into this project.
2. Configure the *Data Alignment* to *Quad Word (8-Byte)* in the *Processor* item of ***32-bit Compiler*** setup of *Project Options*. Because the default *Data Alignment* of Visual C++, Visual Basic and Delphi are in Quad Word (8-Byte), you do not have to change the configuration in those programming language environments.

2.6 Advantech DLL Drivers Programming Considerations

In addition to knowing how to use the Advantech DLL drivers, you should consider some special problems that can occur when you access certain DLL routines. This section briefly describes the nature of the problems. The following sections, which are specific to each language, give the methods for solving the problems.

2.6.1 Buffer Allocation

Allocating memory in a Windows application is much more restrictive than what is normally encountered in a non-Windows application. Windows has its own memory-allocation functions and requires you to allocate all memory through the Windows memory manager. In most cases, you should use these functions rather than the memory-allocation functions normally used by a specific language. Please note that the buffer allocated for handling interrupt function can exceed 64 KB but the buffer allocated for DMA functions cannot be less than 4 KB and cannot exceed 64 KB.

2.6.2 String Passing

When DLLs for Windows routine drivers call for a string that is passed as a parameter, the routines expect a pointer to a null-terminated string. Some languages require special string handling to support this type.

2.6.3 Parameter Passing

You can pass parameters of a procedure or function by value or by reference. Different languages have different default settings. You must pass certain variables by value or by reference to each DLL for Windows functions.

Notice: *The DRV_GetAddress function is for Visual Basic only. In VC++ or Delphi, users can get a pointer or address of a variable. However, in Visual Basic, there is no standard function to get the memory address of a variable. The DLL driver requires an address as a parameter when calling most functions.*

CHAPTER 3

Tutorial

3.1 DLL Driver Introductory Tutorial

This chapter provides an example to demonstrate how to build an application using DLL driver from scratch. The example makes use of the Analog Input Function Group to read one analog value from an analog input channel. A Windows console program, Visual Basic and Delphi are used to build the application and demonstrate the step-by-step procedure. The source code for these programs is also provided. For information about using other function groups or other development tools, please refer to Chapter 2 ***Creating Windows 95/NT Application with DLL Driver*** and Chapter 4 ***Function Overview***. The source code is located in the \advantech\adsapi\examples\tutorial directory on the CD-ROM disc.

The sample program reads an analog input channel from a virtual device and displays the result on the screen. The Advantech DLL driver supports the virtual device named "DEMO BOARD", whose first channel generates a simulated sine wave. By following this example, you will get an overall view about how to program using the DLL driver.

This chapter assumes that you are familiar with the basic concepts of using Visual Basic, Delphi and Visual C++.

3.2 DLL Driver Tutorial for Win32 Console Program

Step 1: Add Demo Board With DEVINST.EXE

1. Go into the Start menu and click on the Device Installation icon in the Advantech drivers folder.

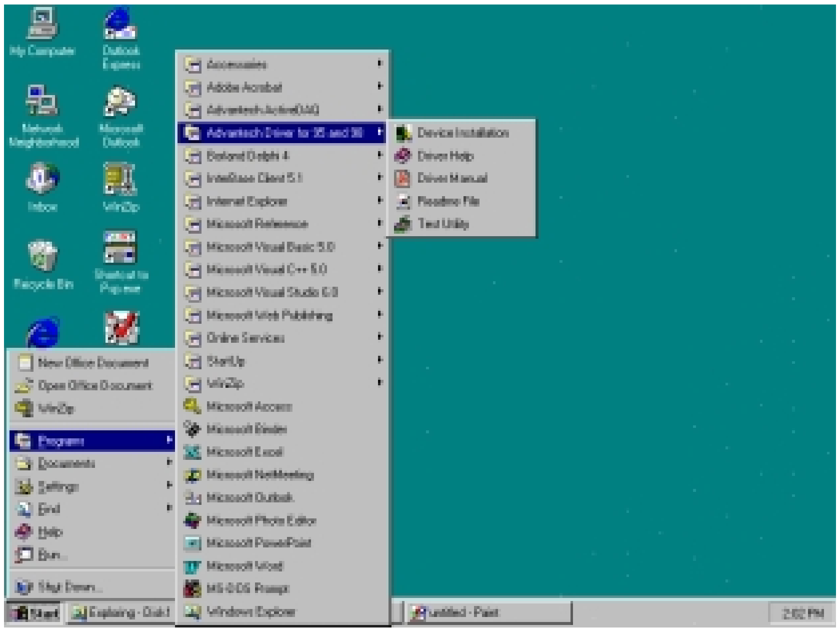


Figure 3-1: Starting the Device Installation Utility

2. The Device Installation Utility will launch as below.

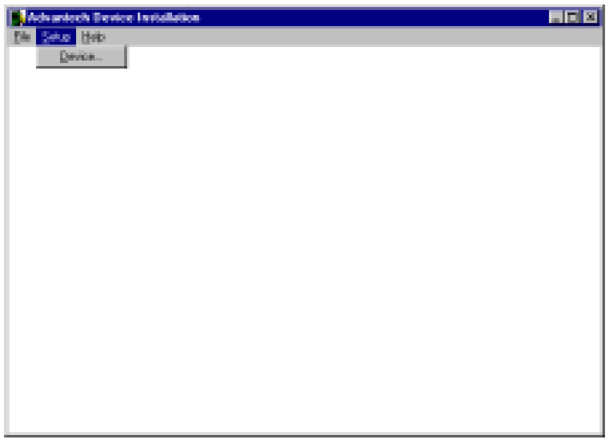


Figure 3-2: Device Installation Utility

3. Select **Setup | Device** from the main menu. A dialog box is displayed:

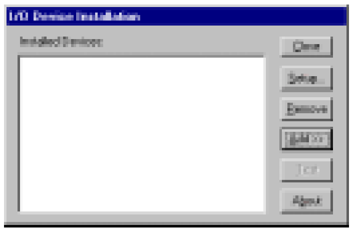


Figure 3-3: I/O Device Installation Dialog Box

4. Press the **Add>>** button and select the *Advantech DEMO Board* item in the *List of Devices* list box.

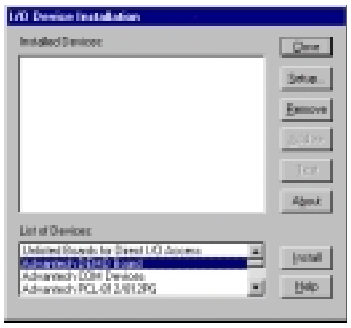


Figure 3-4: Selecting the Advantech DEMO Board Device

5. Press the **Install** button, and a configuration dialog box is displayed as below.

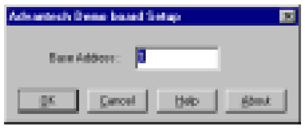


Figure 3-5: Device Configuration Window

6. Use the default value and press the **OK** button. You will see a new entry in the *Installed Devices* list in the *I/O Device Installation* window.

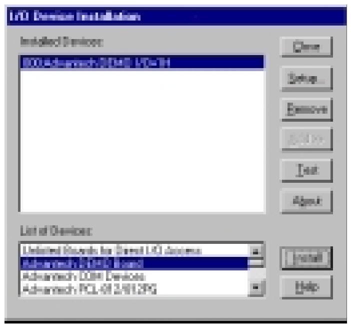


Figure 3-6: I/O Device Installation Dialog Box

7. Press the **Close** button and exit and Device Installation Utility.

Step 2: Write Your Application with DLL Driver

1. Go into the Start menu and click on the Microsoft Visual C++ 5.0 icon in the Microsoft Visual C++ 5.0 folder.

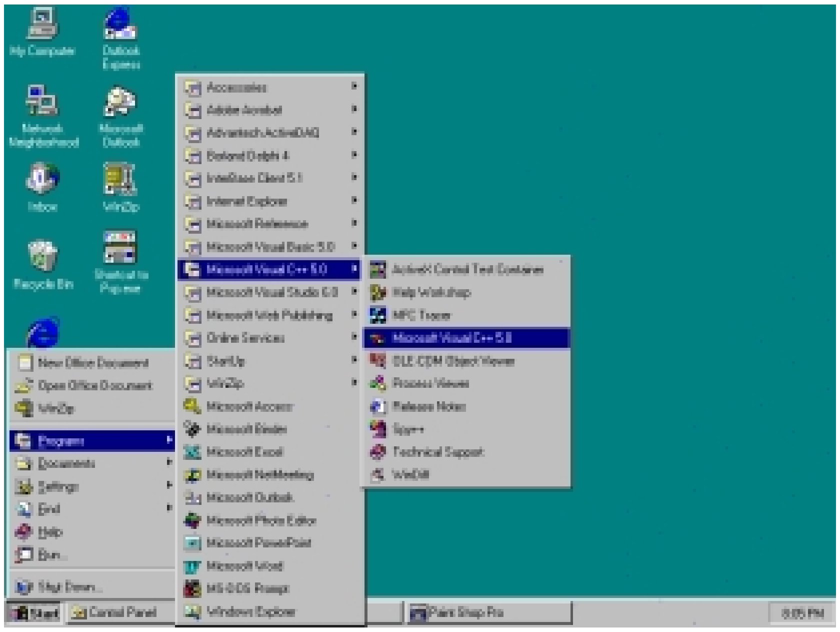
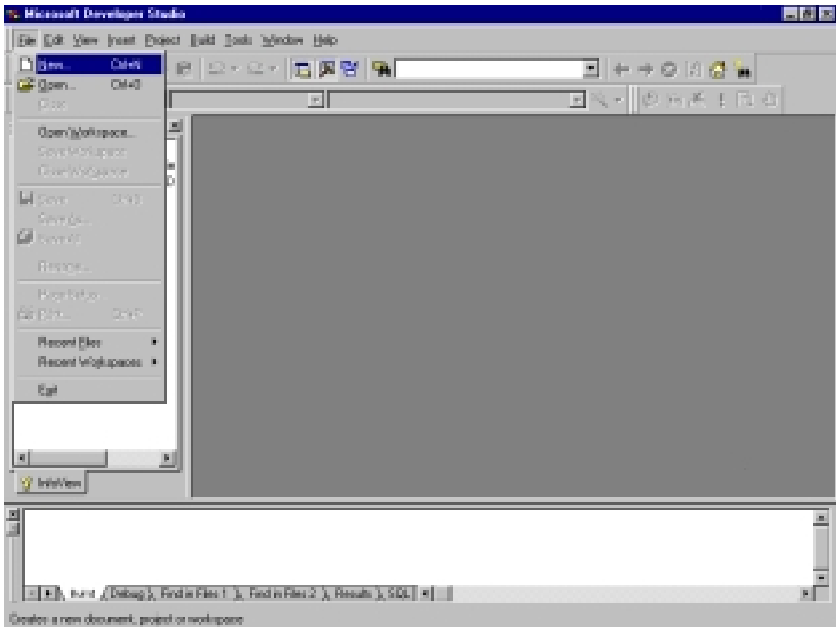


Figure 3-7: Start Microsoft Visual C++

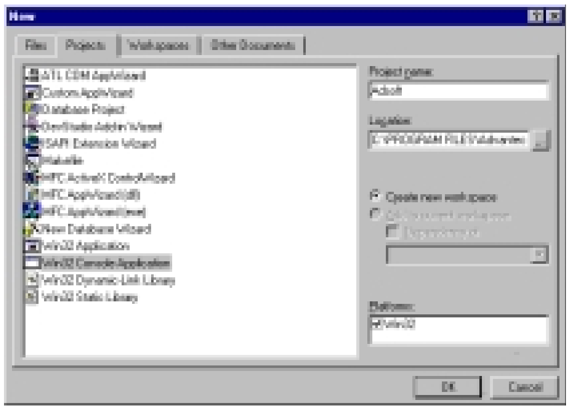
2. Select **File | New**… from the main menu.



Creates a new document, project or workspace

Figure 3-8: Creating a New VC++ Application

3. The following dialog box loads. Click on the *Win32 Console Application* entry in the list and enter “Adsoft” in the *Project* name field. Then press the **OK** button. This generates some skeleton code for you.



Project name: Adsoft

Location: C:\PROGRAM FILES\[illegible]

(●) Create new workspace

Platforms: ☑ Win32

Figure 3-9: Creating a Win32 Console Application in the VC++ App Wizard

4. Add the adsapi32.lib library file and a new file named adsoft.cpp into your project

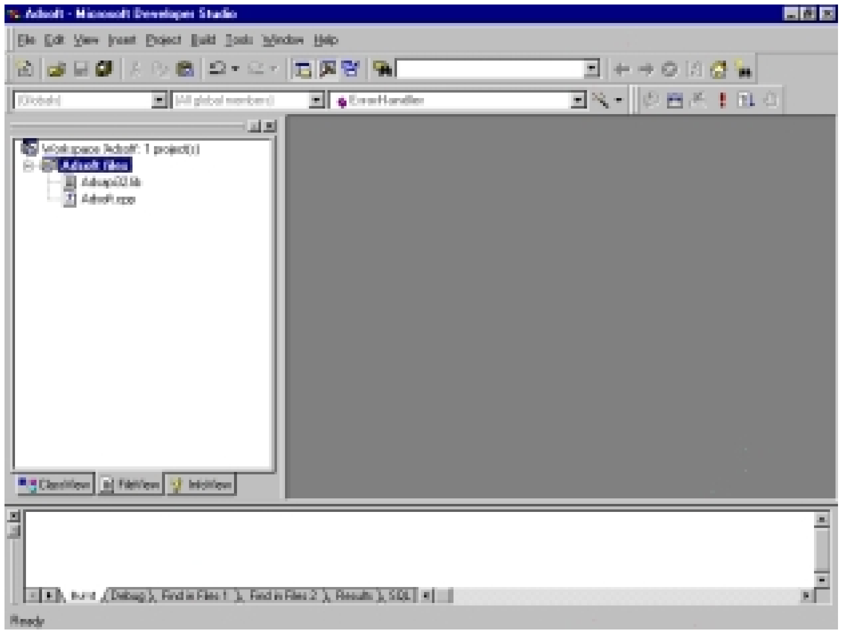


Figure 3-10: Adding the adsapi32.lib Library File and adsoft.cpp to Your Project

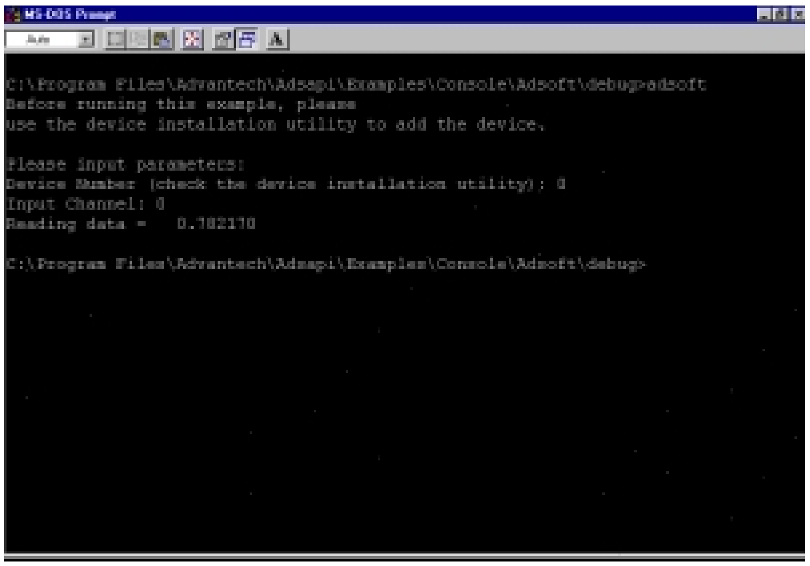
5. Write codes for adsoft.cpp as follows:

```
/*
 **********************************************************************
 * Program        : ADSOFT.CPP
 * Description    : Demo program for analog input function with
*                     software triggering
* Boards Supp.   : PCL-818 series/816/1800/812PG/711B, MIC-2718,
*                     PCM-3718, PCI-1710/1713, PCL-813B, Demo board,
*                     ADAM-4011/4011D/4012/4014D/4018/4018M/
*                     5018/4017/4013/5017/4016
 * APIs used      :  DRV_DeviceOpen,DRV_DeviceClose,DRV_GetErrorMessage*
*                     DRV_AIConfig, DRV_AIVoltageIn
 *
 * Revision       : 1.00
 *
 * Date           : 7/8/1999                   Advantech Co., Ltd.
 ********************************************************************** */
#include <windows.h>
#include <windef.h>
#include <stdio.h>
#include <conio.h>

#define WIN_CONSOLE
#include “..\..\..\include\driver.h”

/******************************
* Local function declaration *
 ******************************/
void ErrorHandler(DWORD dwErrCde);
void ErrorStop(long*, DWORD);

void main()
{
    DWORD  dwErrCde;
    ULONG  lDevNum;
    long   lDriverHandle;
    USHORT usChan;
    float  fVoltage;
    PT_AIVoltageIn ptAIVoltageIn;
    PT_AIConfig ptAIConfig;

    //Step 1: Display hardware and software settings for running this
    //        example
    printf(“Before running this example, please\n”);
    printf(“use the device installation utility to add the device.\n”);

    //Step 2: Input parameters
    printf(“\nPlease input parameters:”);
    printf(“\nDevice Number (check the device installation utility): “);
    scanf(“%d”, &lDevNum);
    printf(“Input Channel: “);
    scanf(“%d”, &usChan);
    //Step 3: Open device
    dwErrCde = DRV_DeviceOpen(lDevNum, &lDriverHandle);
    if (dwErrCde != SUCCESS)
    {
        ErrorHandler(dwErrCde);
        printf(“Program terminated!\n”);
        return ;
    }

    //Step 4: Configure input range
    ptAIConfig.DasChan = usChan;          // channel: 0
    ptAIConfig.DasGain = 0;                // gain code: 0
    dwErrCde = DRV_AIConfig(lDriverHandle, &ptAIConfig);
    if (dwErrCde != SUCCESS)
    {
        ErrorStop(&lDriverHandle, dwErrCde);
        return;
    }

    // Step 5: Read one data
    ptAIVoltageIn.chan = usChan;             // input channel
    ptAIVoltageIn.gain = 0;                   // gain code: refer to
                                                   // manual for range
    ptAIVoltageIn.TrigMode = 0;              // 0: internal trigger,
                                                   // 1: external trigger
    ptAIVoltageIn.voltage = &fVoltage;      // Voltage retrieved

    dwErrCde = DRV_AIVoltageIn(lDriverHandle, &ptAIVoltageIn);
    if (dwErrCde != SUCCESS)
    {
        ErrorStop(&lDriverHandle, dwErrCde);
        return;
    }

    // Step 6: Display reading data
    printf(“Reading data = %10.6f\n”, fVoltage);

    // Step 7: Close device
    dwErrCde = DRV_DeviceClose(&lDriverHandle);
    if (dwErrCde != SUCCESS)
    {
        ErrorStop(&lDriverHandle, dwErrCde);
        return;
    }
}//main

/**********************************************************************
 * Function: ErrorHandler
 * Show the error message for the corresponding error code
 * input:    dwErrCde, IN, Error code
 * return:   none
 **********************************************************************/
```

```
void ErrorHandler(DWORD dwErrCde)
{
    char szErrMsg[180];

    DRV_GetErrorMessage(dwErrCde, szErrMsg);
    printf("\nError(%d): %s\n", dwErrCde & 0xffff, szErrMsg);
}//ErrorHandler

/**********************************************************************
 * Function:   ErrorStop
 * Release all resource and terminate program if error
 * occurs
 * Paramaters: pDrvHandle, IN/OUT, pointer to Driver handle
 * dwErrCde, IN, Error code.
 * return:     none
 *********************************************************************/
void ErrorStop(long *pDrvHandle, DWORD dwErrCde)
{
    //Error message
    ErrorHandler(dwErrCde);
    printf("Program terminated!\n");

    //Close device
    DRV_DeviceClose(pDrvHandle);
    exit(0);
}//ErrorStop
```

Step 3: Test Your Program

1. Click on **Compile** under the **Build** menu to compile your code.
2. Run it under a DOS Prompt.
3. Enter 0 for the device number and 0 for the input channel. The result is shown below:



```
C:\Program Files\Advantech\Adsapi\Examples\Console\Adsoft\debug>adsoft
Before running this example, please
use the device installation utility to add the device.

Please input parameters:
Device Number (check the device installation utility): 0
Input Channel: 0
Reading data =    0.182110

C:\Program Files\Advantech\Adsapi\Examples\Console\Adsoft\debug>
```

Figure 3-11: Running Your Sample Win32 Console Program

3.3 DLL Driver Tutorial for Visual Basic Application

Step 1: Add Demo Board With DEVINST.EXE

1. The same as step 1 in the Win32 Console example.

Step 2: Write Your Application

1. Go into the Start menu and click on the Visual Basic 5.0 icon in the Microsoft Visual Basic folder.

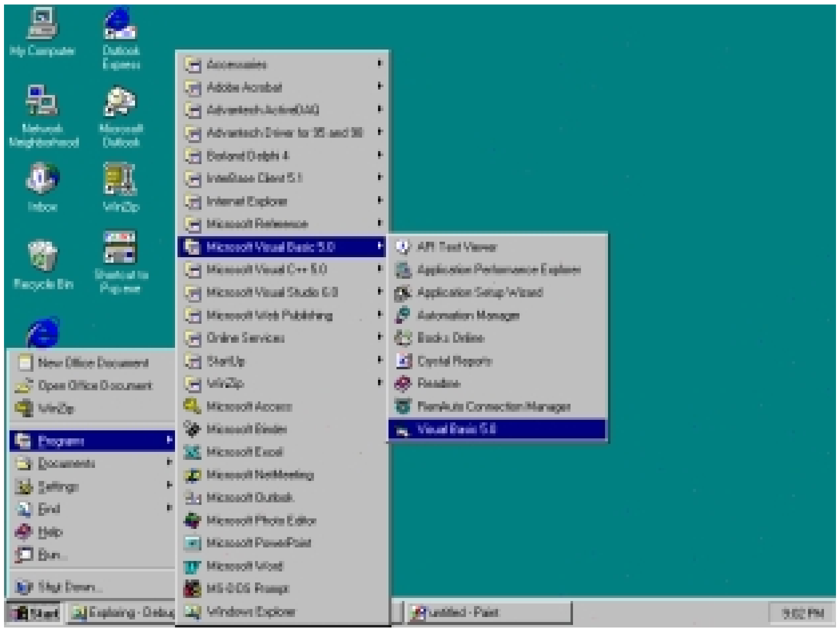


Figure 3-12: Starting Microsoft Visual Basic 5.0

2. The Visual Basic 5.0 development environment will load as follows:

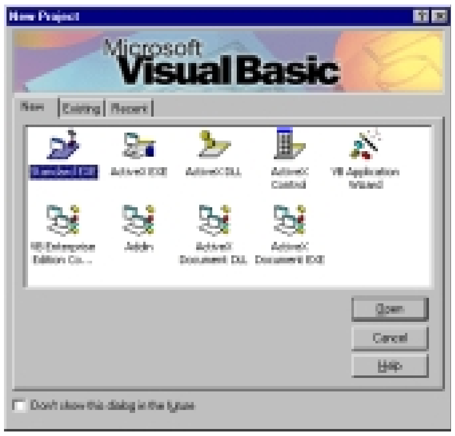


Figure 3-13: Select Standard EXE from the New Project Dialog Box

3. Select the *Standard EXE* icon and press the **Open** button. A new project is created. Then click on the *Project Explorer* in the *View* menu. Add the declaration file, Driver.bas module by clicking on **Add Module** in the **Project** menu. The Driver.bas file is located in the \Advantech\Adsapi\Include directory. Name the adsoft.frm form in the project.

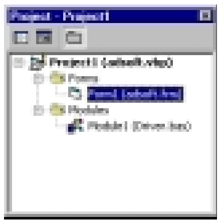


Figure 3-14: Adding the Declaration File driver.bas

4. Design your form: place a Label control on Form1 and enter "Analog Input" as its Caption field. Then place a TextBox control on Form1. Switch to the *Property Window* and enter txtAIValue as its Name property. At last, place a CommandButton control on the form. Enter cmdRead as its Name property, and enter Read as the Caption property. Your form should look similar to the one shown below:

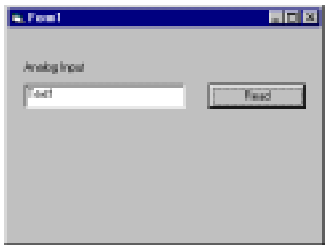


Figure 3-15: Form Design of Form1

5. Write your code for the cmdRead button as below:

```
Private Sub cmdRead_Click()
    Dim ErrCde As Long          ‘ Error code
    Dim szErrMsg As String * 80 ‘ Error string
    Dim DeviceHandle As Long

    Dim AiConfig As PT_AIConfig
    Dim AiVoltageIn As PT_AIVoltageIn

    Dim voltage As Single

    ‘ Step 1: open device
    ErrCde = DRV_DeviceOpen(0, DeviceHandle)    ‘ Make sure device number = 0
    If (ErrCde <> 0) Then
        DRV_GetErrorMessage ErrCde, szErrMsg
        Response = MsgBox(szErrMsg, vbOKOnly, “Error!!”)
        Exit Sub
    End If

    ‘ Step 2: configure input range
    AiConfig.DasChan = 0  ‘ channel: 0
    AiConfig.DasGain = 0  ‘ gain code: 0
    ErrCde = DRV_AIConfig(DeviceHandle, AiConfig)
    If (ErrCde <> 0) Then
        DRV_GetErrorMessage ErrCde, szErrMsg
        Response = MsgBox(szErrMsg, vbOKOnly, “Error!!”)
        Exit Sub
    End If

    ‘ Step 3: read value
    AiVoltageIn.chan = AiConfig.DasChan
    AiVoltageIn.gain = AiConfig.DasGain
    AiVoltageIn.TrigMode = 0
    AiVoltageIn.voltage = DRV_GetAddress(voltage)

    ErrCde = DRV_AIVoltageIn(DeviceHandle, AiVoltageIn)
    If (ErrCde <> 0) Then
      DRV_GetErrorMessage ErrCde, szErrMsg
      Response = MsgBox(szErrMsg, vbOKOnly, “Error!!”)
      Exit Sub
    End If

    ‘ Step 4: display value
    txtAIValue.Text = Format(voltage, “###0.00”)

    ‘ Step 5: close device
    ErrCde = DRV_DeviceClose(DeviceHandle)
    If (ErrCde <> 0) Then
      DRV_GetErrorMessage ErrCde, szErrMsg
      Response = MsgBox(szErrMsg, vbOKOnly, “Error!!”)
    End If

End Sub
```

Notice: The DRV_GetAddress function is for Visual Basic only. In VC++ or Delphi, users can get a pointer or address of a variable. However, in Visual Basic, there is no standard function to get the memory address of a variable. The DLL driver requires an address as a parameter when calling most functions.

Step 3: Test Your Program

1. Press F5 to run the program. Then press the **Read** button in your sample application. The data will appear as below.

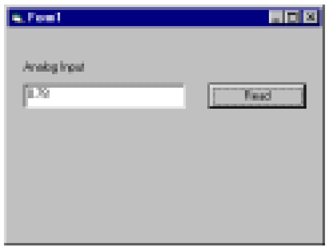


Figure 3-16: Testing the Sample Program

3.4 DLL Driver Tutorial for Delphi Applications

Step 1: Add Demo Board With DEVINST.EXE

1. The same as step 1 for making a Win32 console application.

Step 2: Write Your Application

1. Go into the Start menu and click on the Delphi 4 icon in the Borland Delphi 4 folder.

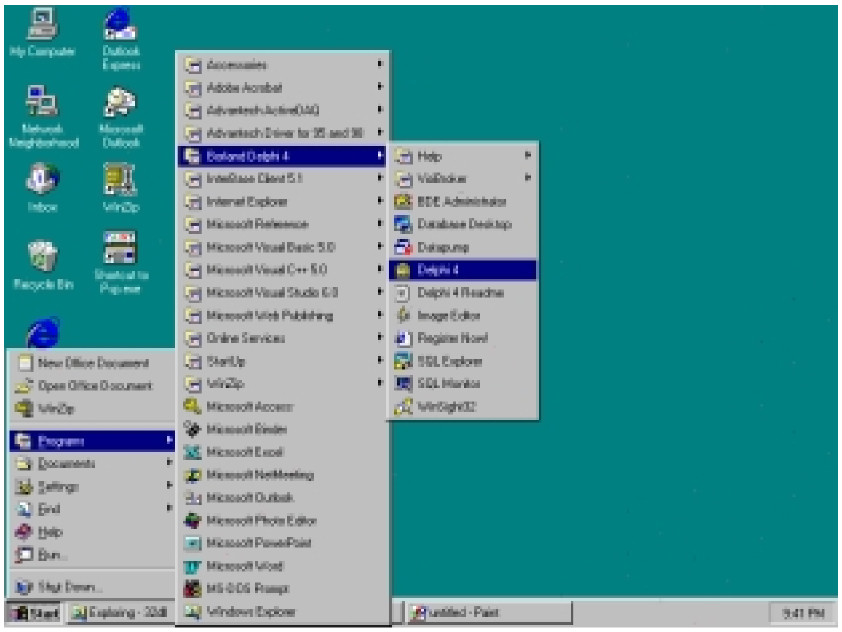
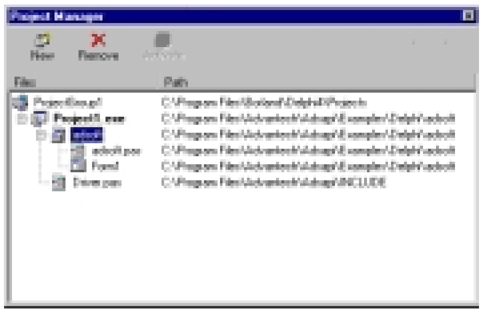


Figure 3-17: Starting Delphi

2. Click on the *Project Manager* in the **View** menu. Add the declaration file, Driver.pas file by clicking the **Add to Project…** button in the **Project** menu. The Driver.pas file is located in \Advantech\Adsapi\Include directory. Name the adsoft.pas unit file for Form1.



Files	Path
ProjectGroup1	C:\Program Files\Borland\Delphi4\Projects
Project1.exe	C:\Program Files\Advantech\Adsapi\Examples\Delphi\adsoft
adsoft	C:\Program Files\Advantech\Adsapi\Examples\Delphi\adsoft
adsoft.pas	C:\Program Files\Advantech\Adsapi\Examples\Delphi\adsoft
Form1	C:\Program Files\Advantech\Adsapi\Examples\Delphi\adsoft
Driver.pas	C:\Program Files\Advantech\Adsapi\INCLUDE

Figure 3-18: Delphi Project Manager

3. Design your form: place a Label control on the Form1 and enter “Analog Input” as its Caption field. Then place an Edit control on the Form1. Switch to the Property Window and enter txtAIValue as its Name property. At last, place a Button control on the form. Enter cmdRead as its Name property, and enter Read as the Caption property. Your form should look similar to the one shown below:

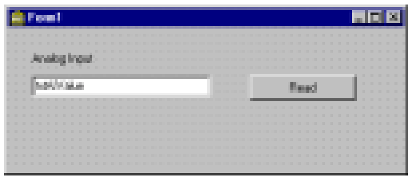


Figure 3-19: Form Design in the Sample Program

4. Write your code for the cmdRead button as below:

```
procedure TForm1.cmdReadClick(Sender: TObject);
var
  voltage                 : Single;
  DeviceHandle            : Longint;
  ErrCde                  : Longint;
  szErrMsg                : string;
  pszErrMsg               : Pchar;
  AiVolIn                 : PT_AIVoltageIn;
  ptAIConfig              : PT_AIConfig;
  Response                : Integer;

begin
  pszErrMsg := @szErrMsg;
  { Step 1: open device }
  ErrCde := DRV_DeviceOpen(0, DeviceHandle);   { device number = 0 }
  If (ErrCde <> 0) Then
    begin
      DRV_GetErrorMessage(ErrCde, pszErrMsg);
      Response := Application.MessageBox(pszErrMsg, ‘Error!!’, MB_OK);
      Exit;
  end;

  { Step 2: configure input range }
  ptAIConfig.DasChan := 0;           { channel: 0 }
  ptAIConfig.DasGain := 0;           { gain code: 0 }
  ErrCde := DRV_AIConfig(DeviceHandle, ptAIConfig);
  If (ErrCde <> 0) Then
  begin
    DRV_GetErrorMessage(ErrCde, pszErrMsg);
    Response := Application.MessageBox(pszErrMsg, ‘Error!!’, MB_OK);
    Exit;
  end;

  { Step 3: read value }
  AiVolIn.chan := ptAIConfig.DasChan;
  AiVolIn.gain := ptAIConfig.DasGain;
  AiVolIn.TrigMode := 0;
  AiVolIn.voltage := @voltage;
  ErrCde := DRV_AIVoltageIn(DeviceHandle, AiVolIn);
  If (ErrCde <> 0) Then
  begin
    DRV_GetErrorMessage(ErrCde, pszErrMsg);
    Response := Application.MessageBox(pszErrMsg, ‘Error!!’, MB_OK);
    Exit;
  end;

  { Step 4: display value }
  txtAIValue.Text := FloatToStrF(voltage, ffFixed, 5, 2);
  { Step 5: close device }
  ErrCde := DRV_DeviceClose(DeviceHandle);
  If (ErrCde <> 0) Then
    begin
      DRV_GetErrorMessage(ErrCde, pszErrMsg);
      Response := Application.MessageBox(pszErrMsg, 'Error!!', MB_OK);
      DRV_DeviceClose(DeviceHandle);
      Exit;
  end;

end;

end.
```

Step 3: Test Your Program

1. Press F9 to run the program. Then press the **Read** button. The data will appear as below.

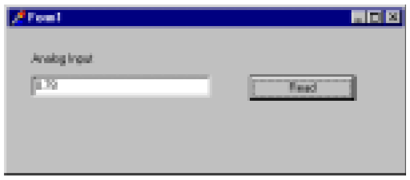


Figure 3-20: Testing the Sample Program

3.5 Advantech DLL Driver Example Programs

The Advantech DLL driver provides the following examples. You can use them to create your own applications easily. Please refer to Chapter 4 ***Function Overview*** for detailed information, such as function description and call flow.

3.5.1 Analog Input With Software Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\adsoft,
\Advantech\Adsapi\Examples\VB\adsoft,
\Advantech\Adsapi\Examples\Delphi\adsoft,
\Advantech\Adsapi\Examples\VC\adsoft,
```

3.5.2 Multiple Channel AI With Software Triggering

Directory:

```
\Advantech\Adsapi\Examples\VB\madsoft
\Advantech\Adsapi\Examples\Delphi\madsoft
\Advantech\Adsapi\Examples\VC\madsoft
```

3.5.3 Analog Input With Interrupt Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\adint
\Advantech\Adsapi\Examples\VB\adint
\Advantech\Adsapi\Examples\Delphi\adint
\Advantech\Adsapi\Examples\VC\adint
```

3.5.4 Analog Input with DMA Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\addma
\Advantech\Adsapi\Examples\VB\addma
\Advantech\Adsapi\Examples\Delphi\addma
\Advantech\Adsapi\Examples\VC\addma
```

3.5.5 Analog Input With Watchdog Triggering

Directory:

```
\Advantech\Adsapi\Examples\VB\cdaddma, cdadint
\Advantech\Adsapi\Examples\Delphi\cdaddma, cdadint
\Advantech\Adsapi\Examples\VC\cdaddma, cdadint
```

3.5.6 Analog Output

Directory:

```
\Advantech\Adsapi\Examples\Console\dasoft
\Advantech\Adsapi\Examples\VB\dasoft
\Advantech\Adsapi\Examples\Delphi\dasoft
\Advantech\Adsapi\Examples\VC\dasoft
```

3.5.7 Synchronous Analog Output

Directory:

```
\Advantech\Adsapi\Examples\Console\dasync
\Advantech\Adsapi\Examples\VB\dasync
\Advantech\Adsapi\Examples\Delphi\dasync
\Advantech\Adsapi\Examples\VC\dasyncc, dasyncv
```

3.5.8 Analog Output with Interrupt Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\daint
\Advantech\Adsapi\Examples\VB\daint
\Advantech\Adsapi\Examples\Delphi\daint
\Advantech\Adsapi\Examples\VC\daint
```

3.5.9 Analog Output with DMA Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\dadma
\Advantech\Adsapi\Examples\VB\dadma
\Advantech\Adsapi\Examples\Delphi\dadma
\Advantech\Adsapi\Examples\VC\dadma
```

3.5.10 Digital Input

Directory:

```
\Advantech\Adsapi\Examples\Console\digin
\Advantech\Adsapi\Examples\VB\digin
\Advantech\Adsapi\Examples\Delphi\digin
\Advantech\Adsapi\Examples\VC\digin
```

3.5.11 Digital Input With Interrupt Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\diint
\Advantech\Adsapi\Examples\VC\diint
```

3.5.12 Digital Input With Pattern Match/Counter/Overflow/Status Change

Directory:

```
\Advantech\Adsapi\Examples\Console\dipattn
\Advantech\Adsapi\Examples\VB\dipattn
\Advantech\Adsapi\Examples\Delphi\dipattn
\Advantech\Adsapi\Examples\VC\dipattn
```

3.5.13 Digital Output

Directory:

```
\Advantech\Adsapi\Examples\Console\digout
\Advantech\Adsapi\Examples\VB\digout
\Advantech\Adsapi\Examples\Delphi\digout
\Advantech\Adsapi\Examples\VC\digout
```

3.5.14 Event Counting

Directory:

```
\Advantech\Adsapi\Examples\Console\counter
\Advantech\Adsapi\Examples\VB\counter
\Advantech\Adsapi\Examples\Delphi\counter
\Advantech\Adsapi\Examples\VC\counter
```

3.5.15 Event Counting With Interrupt Triggering

Directory:

```
\Advantech\Adsapi\Examples\Console\cntint
\Advantech\Adsapi\Examples\VC\cntint
```

3.5.16 Event Counting With Interrupt Triggering

Directory:

```
\Advantech\Adsapi\Examples\VC\qcounter
```

3.5.17 Pulse Output

Directory:

```
\Advantech\Adsapi\Examples\Console\pulse
\Advantech\Adsapi\Examples\VB\pulse
\Advantech\Adsapi\Examples\Delphi\pulse
\Advantech\Adsapi\Examples\VC\pulse
```

3.5.18 PWM Output

Directory:

```
\Advantech\Adsapi\Examples\Console\pulsepwm
\Advantech\Adsapi\Examples\VB\pulsepwm
\Advantech\Adsapi\Examples\Delphi\pulsepwm
\Advantech\Adsapi\Examples\VC\pulsepwm
```

3.5.19 Frequency Measurement

Directory:

```
\Advantech\Adsapi\Examples\Console\freq
\Advantech\Adsapi\Examples\VB\freq
\Advantech\Adsapi\Examples\VC\freq
```

3.5.20 Temperature Measurement

Directory:

```
\Advantech\Adsapi\Examples\Console\thermo
\Advantech\Adsapi\Examples\VB\thermo
\Advantech\Adsapi\Examples\Delphi\thermo
\Advantech\Adsapi\Examples\VC\thermo
```

3.5.21 Alarm

Directory:

```
\Advantech\Adsapi\Examples\VB\alarm
\Advantech\Adsapi\Examples\Delphi\alarm
\Advantech\Adsapi\Examples\VC\alarm
```

3.5.22 Port I/O

Directory:

```
\Advantech\Adsapi\Examples\VB\portio
\Advantech\Adsapi\Examples\Delphi\portio
\Advantech\Adsapi\Examples\VC\portio
```

3.5.23 Communication

Directory:

```
\Advantech\Adsapi\Examples\VC\comm
```

3.5.24 Event Function

Directory:

```
\Advantech\Adsapi\Examples\VB\addma, adint, cdadint, cdaddma, dipattn
\Advantech\Adsapi\Examples\Delphi\addma, adint, cdadint, cdaddma, dipattn,
\Advantech\Adsapi\Examples\VC\addma, adint, cdadint, cdaddma, diint, dipattn,
cntint
```

Note: *We also provide direct I/O examples. You can access them in the Driver CD-ROM disc. They locate in the \dos directory.*

CHAPTER 4

Function Overview

4.1 Introduction

Advantech's 32-bit DLL driver contains a set of functions and associated structures that can be used in various application programs for interfacing with Advantech I/O Device Drivers. The APIs support many development environments and programming languages, including Microsoft Visual C++, Visual Basic and Borland Delphi.

Device drivers are software programs that specify the communications protocol to be used between a peripheral device and the computer. Installing the drivers is necessary to successfully use Advantech I/O products. This documentation describes our drivers' application programming interface (API).

System Overview

An overview of driver functions is shown in Figure 4-1.

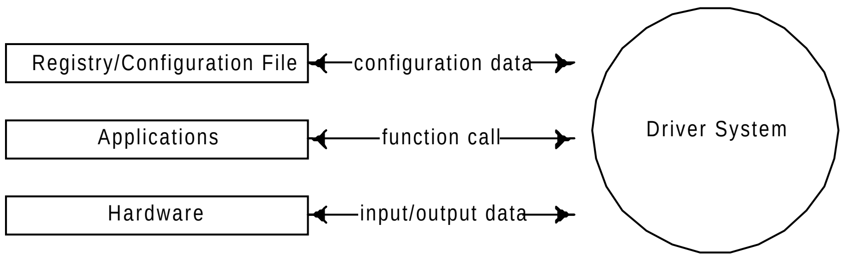


Figure 4-1. Driver System Overview

Component Description

Applications include user programs, configuration utilities and other application software that use the driver.

The driver system prepares for communication between the host computer and Advantech's I/O products. Upon receiving a request from the application, it provides the following services and functions:

1. **Device function**: initializes and configures your hardware and software
2. **Analog input**: converts single and multiple-channel A/D
3. **Analog output**: converts single D/A
4. **Digital I/O**: controls digital I/O for the specified channel.
5. **Port I/O**: controls port I/O.
6. **Counter**: performs event-counting, frequency measurement and pulse output.
7. **Temperature measurement**: measures temperature
8. **Alarm**: operates the alarm
9. **Communication port**: operates the communication port
10. **High speed**: utilize DMA or Interrupt for data acquisition
11. **Hardware**: supports Advantech I/O products.

The Registry/Configuration file stores data about hardware settings. This data will be used for the driver during I/O access.

Interface

Application drivers provide APIs for applications. The application-programming interface (API) is described later in this manual.

Hardware drivers enable communication between peripheral devices and the computer through I/O ports, serial ports and memory. For hardware configuration, please refer to the user's manual that came with your hardware.

Requirements

1 32-bit operating environment, including Windows 95/98/NT.

2 Application programming interface supports the following:

- Microsoft Visual C++ 4.0 (32-bit) or higher for Windows 95/98/NT.
- Microsoft Visual Basic 4.0 (32- bit) or higher for Windows 95/98/NT.
- Borland Delphi 2.0 (32- bit) or higher for Windows 95/98/NT
- Borland C++ 5.0 (32- bit) or higher for Windows 95/98/NT
- Borland C++ Builder 1.0 (32- bit) or higher for Windows 95/98/NT

The driver provides a set of function calls for applications as shown in the following tables.

Device Functions	**Analog Input**	**Analog Output**	**Digital Input/Output**
DRV_DeviceOpen	DRV_AIConfig	DRV_AOConfig	DRV_DioGetConfig
DRV_DeviceClose	DRV_AIGetConfig	DRV_AOBinaryOut	DRV_DioSetPortMode
DRV_DeviceGetFeatures	DRV_AIBinaryIn	DRV_AOVoltageOut	DRV_DioReadPortByte
DRV_GetErrorMessage	DRV_AIScale	DRV_AOScale	DRV_DioWritePortByte
DRV_SelectDevice	DRV_AIVoltageIn		DRV_DioReadBit
DRV_DeviceGetNumOfList	DRV_AIVoltageInExp		DRV_DioWriteBit
DRV_DeviceGetList	DRV_MAIConfig		DRV_DioGetCurrentDOByte
DRV_DeviceGetSubList	DRV_MAIBinaryIn		DRV_DioGetCurrentDOBit
DRV_GetAddress	DRV_MAIVoltageIn		
	DRV_MAIVoltageInExp		

Table 4-1: Device, Analog Input/Output, Digital Input/Output Function Calls

Temp. Measure.	**Port I/O Functions**	**Alarm Functions**	**Counter Functions**
DRV_TCMuxRead	*DRV_WritePortByte*	*DRV_AlarmConfig*	*DRV_CounterEventStart*
	DRV_WritePortWord	*DRV_AlarmEnable*	*DRV_CounterEventRead*
	DRV_ReadPortByte	*DRV_AlarmCheck*	*DRV_CounterFreqStart*
	DRV_ReadPortWord	*DRV_AlarmReset*	*DRV_CounterFreqRead*
	DRV_outp		*DRV_CounterPulseStart*
	DRV_outpw		*DRV_CounterReset*
	DRV_intp		*DRV_QCounterConfig*
	DRV_intpw		*DRV_QCounterConfigSys*
			DRV_QCounterStart
			DRV_QCounterRead

Table 4-2: Temp Measurement, Port I/O, Alarm and Counter Function Calls

High-Speed Function for Analog Input	**High-Speed Function for Analog Output**	**Event Function and Others**
DRV_FAIIntStart	DRV_FAOIntStart	DRV_EnableEvent
DRV_FAIDmaStart	DRV_FAODmaStart	DRV_CheckEvent
DRV_FAIIntScanStart	DRV_FAOLoad	DRV_AllocateDmaBuffer
DRV_FAIDmaScanStart	DRV_FAOCheck	DRV_FreeDmaBuffer
DRV_FAICheck	DRV_FAOStop	DRV_ClearOverrun
DRV_FAITransfer		DRV_TimerCountSetting
DRV_FAIStop		
DRV_FAIDualDmaStart		
DRV_FAIDualDmaScanStart		
DRV_FAIWatchdogConfig		
DRV_FAIIntWatchdogStart		
DRV_FAIDmaWatchdogStart		
DRV_FAIWatchdogCheck		

Table 4-3: High Speed Analog Input, Analog Output, Event and Other Function Calls

4.2 Device Functions

4.2.1 DLL Driver Programming Foundation

The following figure describes the common call flow of the DLL Driver:

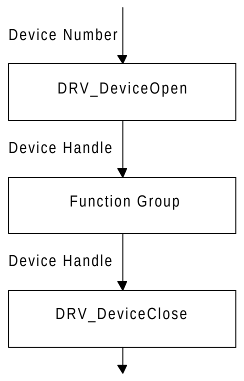


Figure 4-2: DLL Driver Common Call Flow

Device Number (Type: Unsigned Long, Size: 4 bytes)

The Device Number specifies the device that you want to perform the I/O operations. The Device Number is initially defined through configuration using the Device Installation Utility (DEVINST.EXE). The following is the configuration dialog box of the Device Installation Utility. It lists the installed devices.

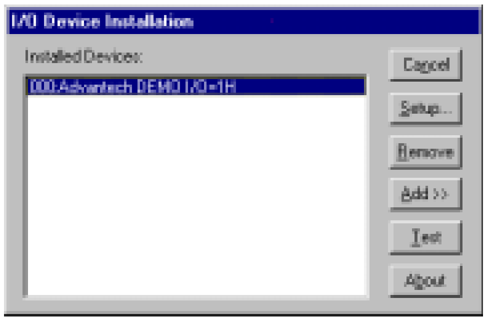


Figure 4-3: Device Installation Utility Configuration Window

For the entry of the device, “000:Advantech DEMO I/O=1H”, the Device Number is equal to 000. You can use assign the Device Number to the DRV_DeviceOpen function directly. Alternatively, you can call DRV_SelectDevice function to generate a dialog box for selecting the desired device, that is Device Number.

DRV_DeviceOpen and DRV_DeviceClose Functions

DRV_DeviceOpen initializes the device specified by Device Number. This function must be called before any other methods that performs I/O operations. DRV_DeviceClose is the counterpart function of the DRV_DeviceOpen function to close the device.

Device Handle (Type: Long, Size: 4 bytes)

Device Handle is set and returned by DRV_DeviceOpen. The subsequent function calls use Device Handle to represent the desired device, instead of Device Number.

Error Code (Type: Unsigned Long, Size: 4 bytes) and DRV_GetErrorMessage

Each driver function returns an error code that indicates whether the function was performed successfully. When a function returns a code that is non-zero, it means that the function failed to perform. You can pass the error code to the DRV_GetErrorMessage function to retrieve its error message.

4.2.2 Other Device Functions

DRV_DeviceGetFeatures

This function accepts a storage address as a function argument, and then returns the device -specific features of the location.

DRV_SelectDevice

This function shows the device list tree dialog box to select the device.

DRV_DeviceGetList

Returns the number and type of installed devices, not including devices attached to COM ports or CAN.

DRV_DeviceGetNumOfList

Returns the number of installed devices.

DRV_DeviceGetSubList

Returns a list of the installed devices attached to a specified COM port or CAN.

DRV_GetAddress

This is only used in Visual Basic. In VC++ or Delphi, users can get a pointer or address of a variable. However, in Visual Basic, there is no standard function to get the memory address of a variable. The DLL drivers require the address as a parameter when calling most functions.

4.3 Analog Input Function Group

The analog input function group performs analog input functions. It can acquire single point data, multiple channel data, and waveform data with interrupt or DMA triggering.

The analog input functions provide four kinds of operation according to the triggering mode and data transfer method.

Parameters

Gain (Type: unsigned short, Size: 2 bytes) and GainList (Type: a pointer to unsigned short array, Size: 4 bytes)

To acquire analog input data from Advantech data acquisition and control (DA & C) cards, you must specify the input range(s). You can specify the input range(s) through the Gain parameter for single channel, or the GainList parameter for multiple channels. Gain and GainList represent physical hardware gain code(s). You can refer to the hardware manual for the mapping of input ranges and gain codes. Alternately, you can use the DRV_GetDeviceFeatures function to get the mapping.

TriggerMode (Type: unsigned short: Size: 2 bytes)

The conversion of analog input can be triggered by internal or external sources. A TriggerMode parameter equal to zero represents internal, and 1 for external.

4.3.1 Software Triggering

These functions trigger the data conversion by software. The driver provides two kinds of functions. One is for single point reading; the other one is for multiple channel reading.

4.3.1.1 Single Point Reading

If you want to sample multiple data periodically by the functions, you can create a software timer to call the functions periodically.

Call Flow

The function call flow is shown below:

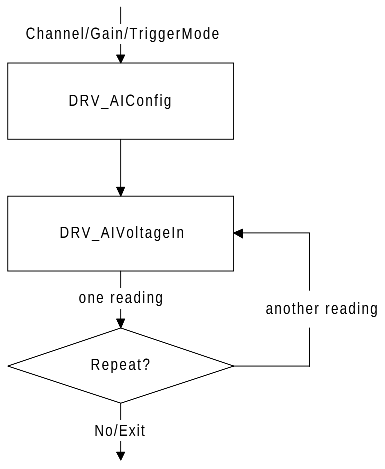


Figure 4-4: Single Point Reading Call Flow Diagram

DRV_AIConfig function configures the input range for the specified channel. If the input range doesn't change at runtime, you just call it once at the beginning. Then use the DRV_AIVoltageIn function to get voltage data repeatedly. The DLL drivers also provides a binary data reading function, DRV_AIBinaryIn. In contrast with the DRV_AIVoltageIn function, it does not convert binary data into voltage data. You can use the DRV_AIScale function to convert it.

Examples Directory

\Advantech\Adsapi\Examples\Console\adsoft

\Advantech\Adsapi\Examples\VB\adsoft

\Advantech\Adsapi\Examples\Delphi\adsoft

\Advantech\Adsapi\Examples\VC\adsoft

Function Description

Demo program for analog input function with software triggering

4.3.1.2 Multiple Channel Scan

The functions for multiple channel sampling are similar to that of single data reading, except the input channel is not limited to one. The function call flow is shown below.

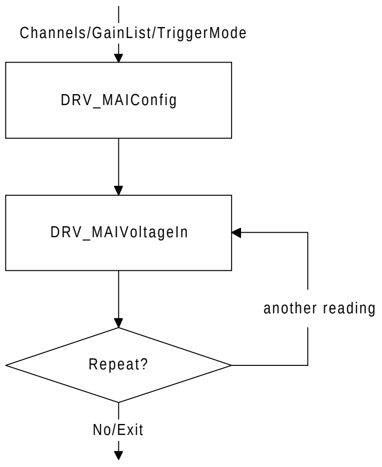


Figure 4-5: Multiple Channel Scan Function Call Flow

The DRV_MAIConfig function configures the input ranges for the specified channels. The DRV_MAIVoltageIn function returns voltage data. Alternately, DRV_MAIBinaryIn returns binary data. You can also use the DRV_AIScale function repeatedly to convert them.

4.3.1.3 Other Functions

DRV_AIVoltageInExp

Reads from an analog input channel that is attached to an expansion board and returns the voltage.

DRV_MAIVoltageInExp

Performs multiple-channel analog inputs on the expansion board and returns the voltages.

Examples Directory

\Advantech\Adsapi\Examples\VB\madsoft

\Advantech\Adsapi\Examples\Delphi\madsoft

\Advantech\Adsapi\Examples\VC\madsoft

Function Description

Demo program for multiple channel analog input function with software triggering.

4.3.2 Waveform Data Reading

The analog input function group provides three kinds of waveform data acquisition. They are interrupt triggering, DMA triggering, and watchdog triggering. For single point data reading functions, it samples data by software triggering. However, waveform data reading utilizes the on-board pacer to trigger the sampling operation and acknowledge the driver through a hardware interrupt.

4.3.2.1 Waveform Data Acquisition Operation Theory

This kind of data transfer is performed in the background. It thus uses less CPU time and speeds the transfer rate. These functions are used for larger amounts of data transfer at higher rates. The driver uses double-buffering techniques for continuous, uninterrupted data transfer of large amounts of data.

Single-buffered Input

In single-buffered input operations, a fixed number of samples are acquired at a specified rate and transferred into computer memory. After the data is stored into the memory buffer, the computer can analyze, display, or store the data to the hard disk for later processing. Single-buffered output operations output a fixed number of samples from computer memory at a specified rate. After the data is output, the buffer can be updated with new or freed data.

Single-buffered operations are relatively simple to implement and can usually take advantage of the full hardware speed of the data acquisition board. The major disadvantage of single-buffered operation is that the amount of data at any one time is limited to the amount of free memory available in the computer and the available count in the DMA count register (i.e., 64K).

Double-buffered Input

In double-buffered operations, the data buffer is configured as a circular buffer. The data acquisition board fills the circular buffer with data. When the end of the buffer is reached, the board returns to the beginning of the buffer and fills it with data again. The process continues until it is interrupted by a hardware error or cleared by a function call.

Unlike single-buffered operations, double-buffered operations reuse the same buffer and are therefore able to input an infinite number data points without requiring an infinite amount of memory. However, in order for double buffering to be useful, there must be a means by which to access the data for updating, storage, and processing.

The driver logically divides the circular buffer into two equal halves. By dividing the buffer into two halves, the driver can coordinate user access to the data buffer with the data acquisition board. The double-buffered input operation begins when the data acquisition board starts writing data into the first half of the circular buffer. After the board begins writing to the second half of the circular buffer and before the board writes to the first half of the circular buffer again, the user needs to copy the data from the first half into the transfer buffer by calling a function. The user can then store the data in the transfer block to disk or process it according to the needs of his application. After the input board has filled the second half of the circular buffer, the board returns to the first half buffer and overwrites the old data. Users can now copy the second half of the circular buffer to the transfer buffer. The data in the transfer buffer is again available for use by the user's application. The process can be repeated endlessly to provide a continuous stream of data to user applications.

The double-buffered coordination scheme is not flawless. An application might experience two possible problems with double-buffered input. The first is the possibility of the data acquisition board overwriting data before the user copies it to the transfer buffer. In this situation, the driver returns an overrun warning. Subsequent transfers will not return the warning as long as they keep pace with the data acquisition board. The second potential problem occurs when an input board overwrites data that the user is simultaneously copying to the transfer buffer. The driver will return an overrun warning.

Background Status

For data acquisition, the user needs to check the status of the operation and retrieve the data for avoiding the data overrun. Users may call the DRV_AICheck function frequently to check the status and DRV_AITransfer to copy the data. The driver provides another method to actively inform the user's application. User calls DRV_EnableEvent with enabling message sending. The driver will then send messages when the hardware interrupt occurs. Please refer to the Event Message Functions section for further details.

Parameters

Cyclic (Type: Unsigned Short, Size: 2 bytes)

To acquire waveform data, the analog input function group provides single shot (non-cyclic) and continuous acquisition (cyclic). You can set the Cyclic parameter to 1 for continuous operation, or 0 for one shot operation.

SampleRate (Type: Floating Point, Size: 4 bytes)

The SampleRate parameter specifies the rate for sampling one data in Hz. The driver uses it to program the on-board pacer.

Count (Type: Unsigned Long, Size: 4 bytes)

The Count parameter specifies the number of samples acquired in a one-shot acquisition or continuous acquisition. The driver allocates the same size of the buffer to store the acquired data.

Note: *The Count parameter ranges from 1 to 65535. For continuous acquisition, it affects the sampling rate. If the sampling rate is higher, the Count parameter should be larger.*

4.3.2.2 Interrupt Triggering

There are two kinds of interrupt operations for analog input. One generates a hardware interrupt for each conversion. The other one keeps the conversion data in FIFO, then generates a hardware interrupt for half-full of FIFO, or full of FIFO. It depends on the hardware.

Parameters

IntrCount (Type: Unsigned Short, Size: 2 bytes)
The IntrCount parameter determines how many conversions generate a hardware interrupt. It depends on the hardware.

Call Flow

The following shows the function call flow for single channel reading.

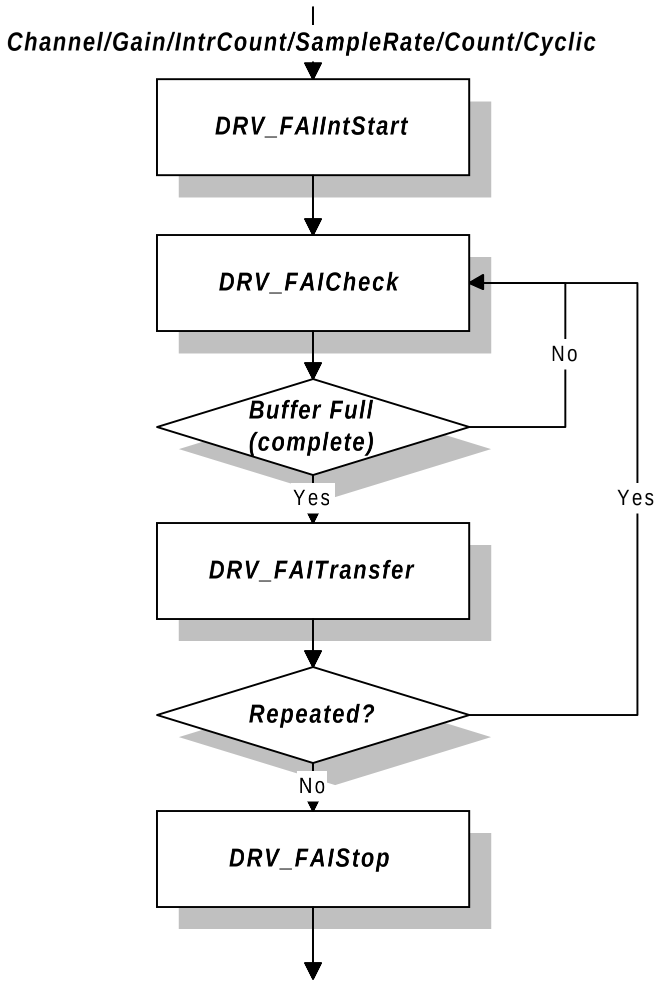


Figure 4-6: Single Channel Reading Function Call Flow

The DRV_FAIIntStart function starts the analog input operation with interrupt triggering. It runs in the background. You can use the DRV_FAICheck function to check the status of the operation. Meanwhile you can use DRV_FAITransfer function to retrieve the data. Then you can stop the operation by DRV_FAIStop function when the conversion is complete or at any other time.

Note: Besides single data reading, the driver also provides DRV_FAIIntScanStart function for multiple channel reading. For this kind of operation, you should set the buffer size, that is the Count parameter, to be a multiple of the number of channels. Otherwise, for the cyclic mode, the first data in the circular buffer may not be for the start channel.

Examples Directory

\Advantech\Adsapi\Examples\Console\adint

\Advantech\Adsapi\Examples\VB\adint

\Advantech\Adsapi\Examples\Delphi\adint

\Advantech\Adsapi\Examples\VC\adint

Function Description

Demo program for analog input function with Interrupt triggering

4.3.2.3 DMA Triggering

There are two kinds of DMA triggering method. One is single DMA triggering; the other one is dual DMA triggering. Dual DMA triggering utilities two DMA channels for data transfer. It depends on hardware. PCL-1800 supports dual DMA triggering.

DMA Buffer

Because of the AT and Micro Channel bus architectures, the DMA region is limited to lie in the 1 MB of physical memory and must be continuous and fixed. The Windows API does not support this kind of buffer allocation. Therefore, driver provides two function calls, DRV_AllocateDMABuffer and DRV_FreeDMABuffer, to allocate DMA buffer and free the allocated DMA buffer.

Note: The DMA buffer size must exceed 4K bytes.

Call Flow

The following shows the call flow for single channel reading.

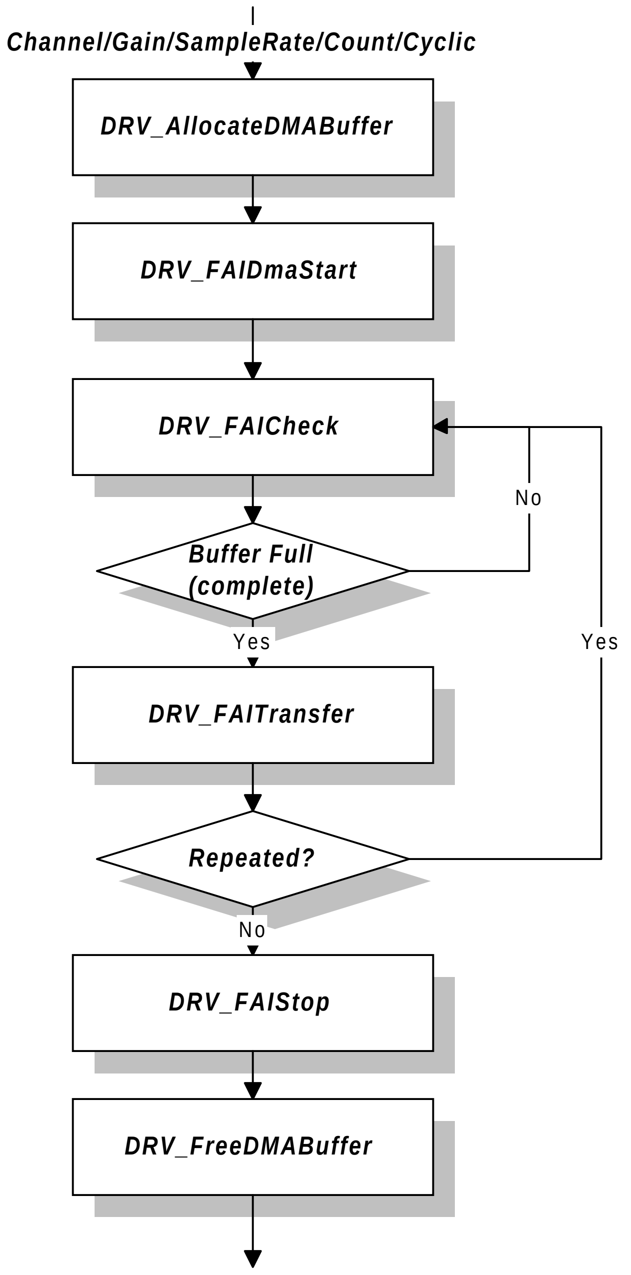


Figure 4-7: Single Channel Reading Call Flow

This is almost the same as interrupt triggering except the DRV_FAIIntStart function starts the analog input operation with DMA triggering. In addition, it uses the DRV_AllocateDMABuffer function to allocate the DMA buffer before starting the DMA operation and uses DRV_FreeDMABuffer to free the DMA buffer after finishing the DMA operation.

Note: *Besides single data reading, the driver also provides the DRV_FAIDmaScanStart function for multiple channel reading. For this kind of operation, you should set the buffer size to be a multiple of the number of channels. Otherwise, for the cyclic mode, the first data in the circular buffer may not be for the start channel.*

For dual DMA triggering, it is similar to the single DMA triggering except uses DRV_FAIDualDmaStart or DRV_FAIDualDmaScanStart to start the dual DMA triggering for single channel reading or multiple channel reading. Besides, you have to allocate two DMA buffers. This is only supported in the PCL-1800 model.

Examples Directory

\Advantech\Adsapi\Examples\Console\addma

\Advantech\Adsapi\Examples\VB\addma

\Advantech\Adsapi\Examples\Delphi\addma

\Advantech\Adsapi\Examples\VC\addma

Function Description

Demo program for analog input function with DMA triggering

4.3.2.4 Watchdog Triggering

This triggering mode is only supported in the PCL-1800. For data acquisition with the watchdog function (analog comparator), it acquires data and compares it against the triggering levels and conditions in the watchdog. It has four triggering modes: free-run, pre-trigger, post-trigger and position-trigger. Free-run means that it ignores the level triggering. For pre-trigger, it acquires and stores data in the circular buffer until the watchdog is triggered. On the other hand, post-trigger mode acquires data after the watchdog is triggered. For position-trigger, it acquires data before and after the watchdog is triggered. This mode needs two buffers to store the data. The data before the watchdog is triggered, including the triggering data, is stored in the first buffer. The data after the watchdog is triggered, is stored in the second buffer. These modes can be combined in the cyclic mode. The relationship is as follows:

	Buffer A	**Buffer B**	**Transfer Mode**
Pre-trigger	circular buffer	not used	DMA or Interrupt
Post-trigger	circular buffer or linear buffer specified by cyclic mode	not used	DMA or Interrupt
Position-trigger	circular buffer	circular buffer or linear buffer specified by cyclic mode	DMA only

Table 4-4: Watchdog Triggering Relationships

Parameters

CondList (Type: a Pointer to Unsigned Short Array, Size: 4 bytes)

The CondList parameter is a pointer to the condition array. It specifies the triggering mode(s) for the scanning channel(s). The triggering modes include free-run (0), pre-trigger (1), post-trigger (2), and position-trigger (3).

LevelList (Type: a pointer to TRIGLEVEL Structure Array, Size: 4 bytes)

The LevelList parameter is a pointer to the level array. It specifies low and high limits for the scanning channel(s).

Note: The TRIGLEVEL data structure is defined as below:

```
typedef struct tagTRIGLEVEL
{
     FLOAT fLow;
     FLOAT fHigh;
} TRIGLEVEL;
```

Call Flow

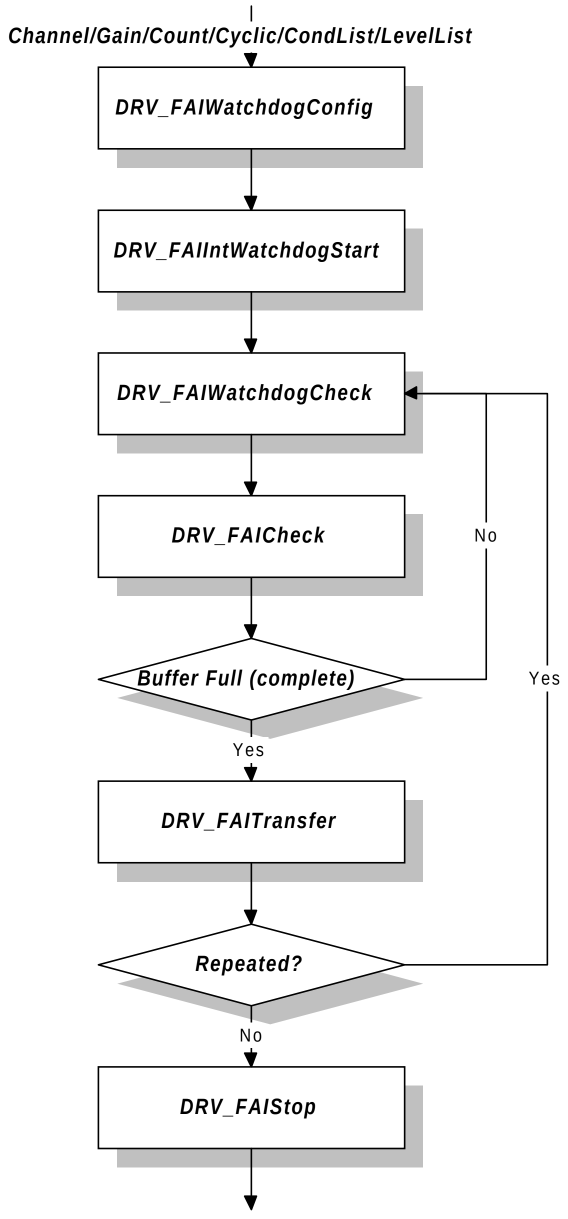


Figure 4-8: Watchdog Triggering Call Flow

The DRV_FAIWatchdogConfig function configures the triggering conditions and levels for each channel. The DRV_FAIIntWatchdogStart function starts the watchdog triggering. It then runs in the background. You can use the DRV_FAIWatchdogCheck function to check if the watchdog condition is matched with the triggered channel. After the watchdog condition is matched, you can use DRV_FAICheck to check the conversion status and DRV_FAITransfer to retrieve the data. The rest is the same as interrupt triggering.

Note: Besides interrupt transfer, the driver also provides DRV_FAIDmaWatchdogStart function for DMA transfer.

Example Directory

\Advantech\Adsapi\Examples\VB\cdaddma

\Advantech\Adsapi\Examples\Delphi\cdaddma

\Advantech\Adsapi\Examples\VC\cdaddma

\Advantech\Adsapi\Examples\VB\cdadint

\Advantech\Adsapi\Examples\Delphi\cdadint

\Advantech\Adsapi\Examples\VC\cdadint

Function Description

Demo program for analog input function with Watchdog triggering

4.3.3 Performance

Interrupt latency in Windows can impose performance limitations on data acquisition. Interrupt latency is the delay between the time hardware asserts an interrupt and when the interrupt service routine is activated. In DOS, the interrupt latency is minimal because the hardware transfers control directly to the interrupt service routine. In Windows, however, system software transfers control to the interrupt service routine, imposing a software delay. The interrupt latency limits the performance of the interrupt data transfer. It also can slow data acquisition that uses DMA when DMA reprogramming is required, because it can cause significant pauses between data transfer requests from the DMA controller. The driver may have to reprogram the DMA controller for a conversion count larger than 64 KB. Pauses during high-speed input operations can cause acquisition boards to miss or overwrite data points.

To eliminate the interrupt latency, the driver does not reprogram the DMA for a conversion count larger than 64 KB. It utilizes the double-buffering technique. In addition, some data acquisition boards provide FIFO interrupt to reduce the frequency of the hardware interrupt. The data is stored in FIFO on board. After some specified conversion count, the input board generates an interrupt to inform the driver to retrieve the data from the FIFO. The data retrieving does not disturb the data acquisition.

4.4 Analog Output Function Group

The analog output function group performs analog output functions. It includes single point data output, waveform data output, and synchronous output.

4.4.1 Single Point Output

Call Flow

You only have to call the DRV_AOVoltageOut function for single point analog output.

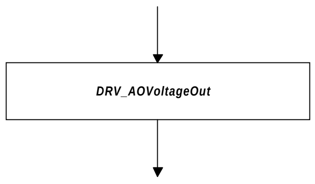


Figure 4-9: Single Point Output Call Flow

The DRV_AOVoltageOut function accepts a floating-point voltage value, scales it to the proper binary number, and writes that number to an analog output channel to change the output voltage. The output range is based on the settings of the DA Reference Voltage in the Device Installation Utility. You can change the output range by using the DRV_AOConfig function at runtime.

The DLL driver also provides a binary data output function, DRV_AOBinaryOut. It accepts a binary value and writes it to an analog output channel. You can use DRV_AOScale function to convert the desired analog output value into a binary value. You can then use DRV_AOBinaryOut function to output the value.

Examples Directory

\Advantech\Adsapi\Examples\Console\dasoft

\Advantech\Adsapi\Examples\VB\dasoft

\Advantech\Adsapi\Examples\Delphi\dasoft

\Advantech\Adsapi\Examples\VC\dasoft

Function Description

Demo program for analog output function with software triggering

4.4.2 Multiple Channel Synchronous Output

These functions are only supported in the PCI-1720.

Call Flow

The call flow of multiple channel synchronous output is shown below.

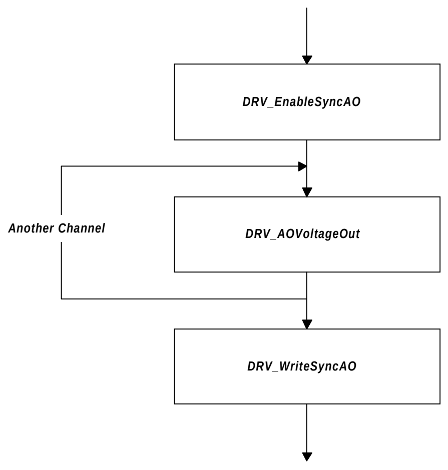


Figure 4-10: Multiple Channel Synchronous Output Call Flow

The DRV_EnableSyncAO function enables the synchronized output operation. You can then call the DRV_AOVoltageOut function repeatedly to set the voltage value for each output channel. Finally, write the output values to all channels synchronously by using DRV_WriteSyncAO function. Besides the DRV_AOVoltageOut function, you can use DRV_AOCurrentOut function for current output.

Examples Directory

\Advantech\Adsapi\Examples\Console\dasync

\Advantech\Adsapi\Examples\VB\dasync

\Advantech\Adsapi\Examples\Delphi\dasync

\Advantech\Adsapi\Examples\VC\dasyncc

\Advantech\Adsapi\Examples\VC\dasyncv

Function Description

Demo program for synchronous analog output function

4.4.3 Waveform Analog Output

The analog output function group provides two kinds of waveform data ouput. They are interrupt triggering and DMA triggering. The waveform analog output uses the on-board pacer to trigger the output operation and acknowledge the driver through a hardware interrupt.

Background Status

For analog output, a user needs to check the status of the operation and load new data for cyclic mode. The user may frequently call the DRV_FAOCheck function to check the status and DRV_FAOLoad to load the new data. The driver provides another method to actively inform the user's application. The user calls DRV_EnableEvent to enable message sending. The driver will then send messages when the hardware interrupt occurs. Please refer to Event Message Functions section for details.

Parameters

Cyclic (Type: Unsigned Short, Size: 2 bytes)

To output waveform data, the analog input function group provides non-cyclic and continuous (cyclic) output. You can set the Cyclic parameter to 1 for continuous operation, or 0 for non-cyclic operation.

SampleRate (Type: Floating Point, Size: 4 bytes)
The SampeRate parameter specifies the rate to output one data item in Hz. The driver uses it to program the on-board pacer.

Count (Type: Unsigned Long, Size: 4 bytes)
The Count parameter specifies the number of outputs. It ranges from 1 to 65535

TrigSrc (Type: unsigned short: Size: 2 bytes)
The operation of analog output can be triggered by internal or external sources. The TrigSrc parameter equals zero to represent internal, and 1 for external.

4.4.3.1 Interrupt Triggering

Call Flow

The call flow for analog output with interrupt triggering is shown below.

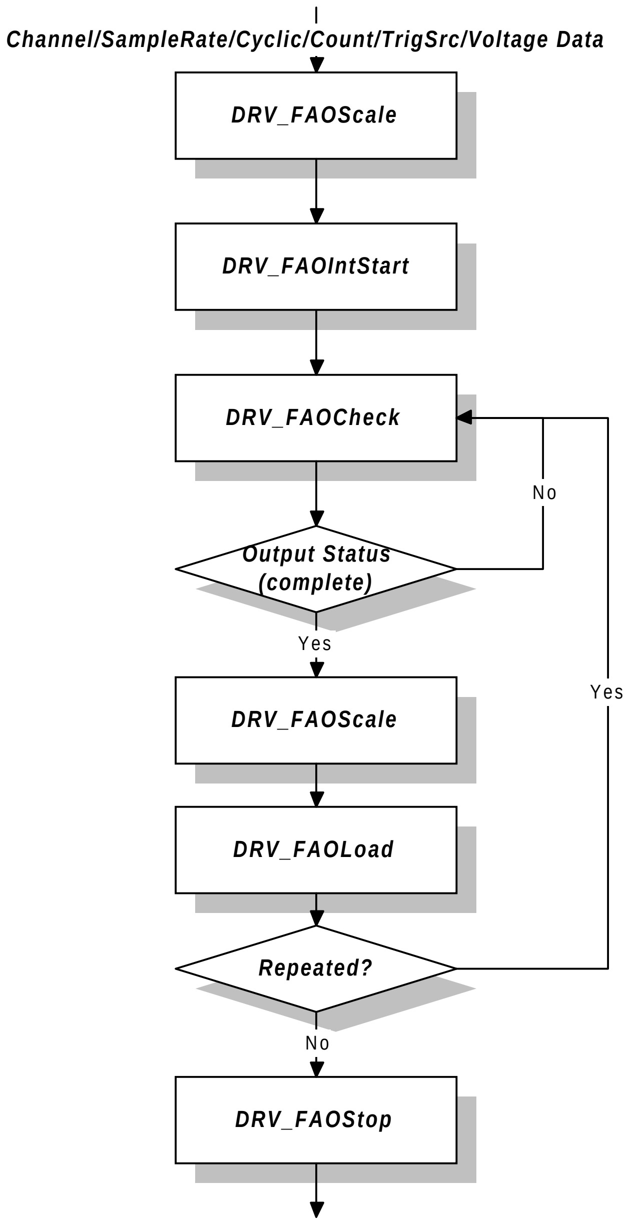


Figure 4-11: Interrupt Triggering Function Call Flow

The DRV_FAOIntStart function starts the operation of analog output with interrupt triggering. Before you call it, you must scale your voltage output data into binary data by the DRV_FAOScale function. When the operation is running, you can use the DRV_FAOCheck function to check the background status. Meanwhile, you can reload the output data by using the DRV_FAOScale and DRV_FAOLoad functions when the output is complete. If you have scaled the output data to binary data, you can skip the DRV_FAOScale function. After all output operation is complete, you can call DRV_FAOStop function to terminate the operation.

Examples Directory

\Advantech\Adsapi\Examples\Console\daint

\Advantech\Adsapi\Examples\VB\daint

\Advantech\Adsapi\Examples\Delphi\daint

\Advantech\Adsapi\Examples\VC\daint

Function Description

Demo program for analog output function with interrupt triggering.

4.4.3.2 DMA Triggering

Call Flow

The call flow for analog output with DMA triggering is shown below.

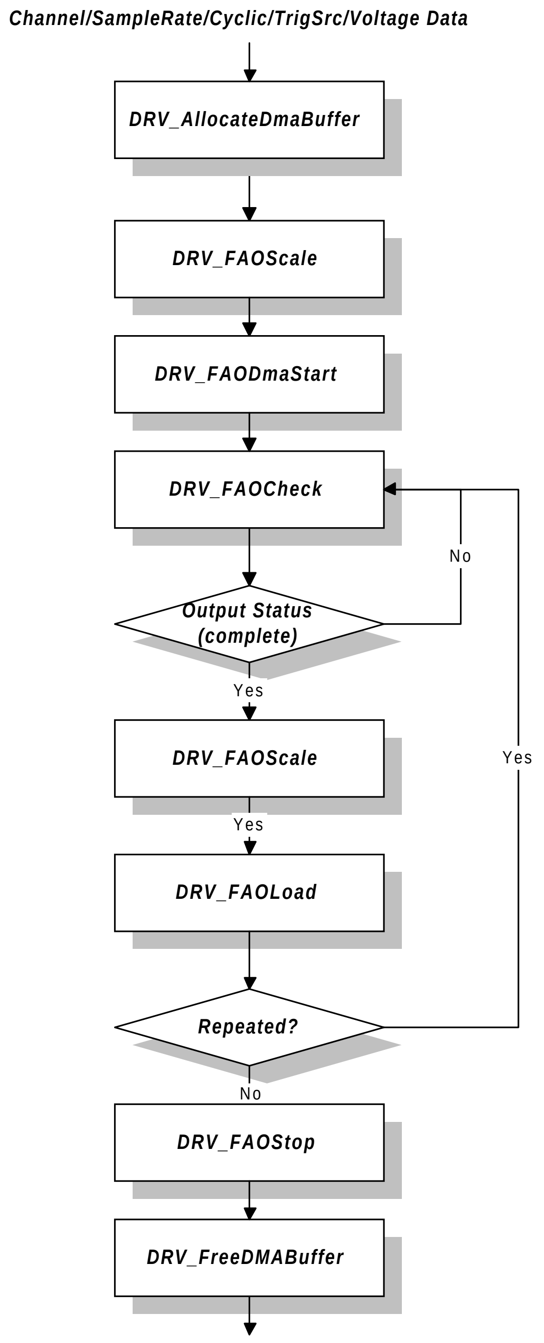


Figure 4-12: DMA Triggering Function Call Flow

The call flow is the same as for interrupt triggering, except that the operation is started by the DRV_FAODmaStart function. In addition, you must call DRV_AllocateDMABuffer to allocate the DMA buffer and DRV_FreeDMABuffer to free the DMA buffer.

Examples Directory

\Advantech\Adsapi\Examples\Console\dadma

\Advantech\Adsapi\Examples\VB\dadma

\Advantech\Adsapi\Examples\Delphi\dadma

\Advantech\Adsapi\Examples\VC\dadma

Function Description

Demo program for analog output function with DMA triggering.

4.5 Digital Input/Output Function Group

The Digital Input/Output function group performs digital input and output operations. The digital input/output lines (bits) on each data acquisition device are grouped into logical units called ports. Each port has eight lines or bits. For example, the port 1/bit 3 specifies the eleventh bit on the data acquisition device. The DLL drivers provides bit and port (byte) functions

The digital I/O port of some data acquisition devices (e.g., PCL-722/724/731) can be configured for input or output. You can use the DRV_DioSetPortMode function to configure the specified port for input or output. In addition, you can use the DRV_DioGetConfig function to read the configuration.

4.5.1 Digital Input Functions

The digital input functions perform digital input operations. The DLL drivers support digital input with software triggering, and digital input with interrupt.

4.5.1.1 Software Triggering

Call Flow

You just call DRV_DioReadBit function to read the state from the specified bit.

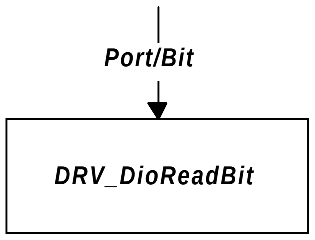


Figure 4-13: DI Software Triggering Call Flow

In addition to the DRV_DioReadBit function, the DLL drivers also provide the DRV_DioReadPortByte function to read a byte value from a port.

Examples Directory

\Advantech\Adsapi\Examples\Console\digin

\Advantech\Adsapi\Examples\VB\digin

\Advantech\Adsapi\Examples\Delphi\digin

\Advantech\Adsapi\Examples\VC\digin

Function Description

Demo program for digital input function

4.5.1.2 Interrupt Triggering

The digital input functions with interrupt triggering allow you to monitor the status of the digital input line. When the state changes from low to high or/and from high to low, it would acknowledge the driver through a hardware interrupt. You don’t have to poll the digital input line periodically.

Call Flow

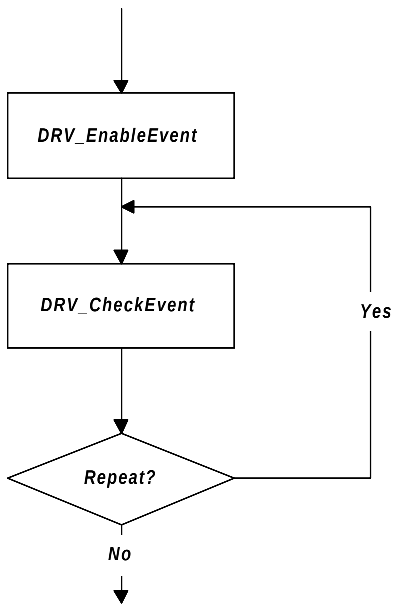


Figure 4-14: Interrupt Triggering Call Flow

The DRV_EnableEvent function enables and starts digital input with interrupt triggering. You can then use the DRV_CheckEvent function to check the background status. The DRV_CheckEvent function returns the interrupt event type when the state changes. It also allows you to set a time-out interval.. Please refer to the event functions for details.

Examples Directory

\Advantech\Adsapi\Examples\Console\diint

\Advantech\Adsapi\Examples\VC\diint

Function Description

Demo program for digital input function with interrupt triggering.

4.5.1.3 Pattern Matched/Status Change/Counter/Filter

The digital input functions with pattern matched/status change/counter/filter triggering allow you to monitor the status of the digital input lines. When one of the conditions is matched, it will acknowledge the driver through a hardware interrupt. You do not have to periodically poll the digital input line. These functions are only supported in PCI-1753/1760.

Call Flow

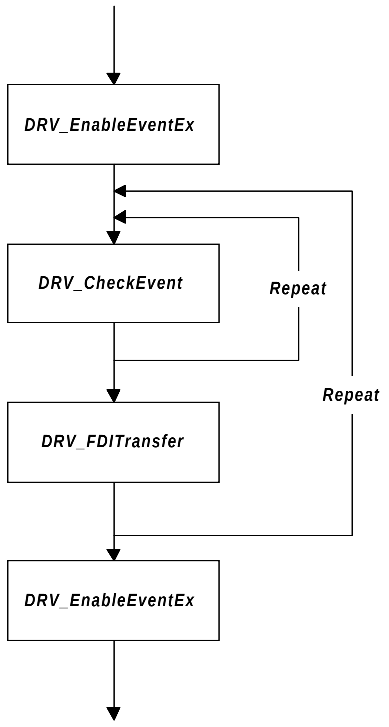


Figure 4-15: Pattern Matched/Status Change/Counter/Filter Function Call Flow

The DRV_EnableEventEx function enables and starts the monitoring of the pattern match/status change/filter/counter operation. You can then use the DRV_CheckEvent function to check the background status. The DRV_CheckEvent function returns the interrupt event type when one of the conditions is matched. You can also use the DRV_FDITransfer function to retrieve the value of the input lines. Finally, you use the DRV_EnableEventEx function to disable and stop the operation.

Examples Directory

\Advantech\Adsapi\Examples\Console\dipattn

\Advantech\Adsapi\Examples\VB\dipattn

\Advantech\Adsapi\Examples\Delphi\dipattn

\Advantech\Adsapi\Examples\VC\dipattn

Function Description

Demo program for digital input function with pattern match.

4.5.2 Digital Output Functions

The digital output functions perform digital output operations.

Call Flow

You simply call the DRV_DioWriteBit function to set the state to the specified bit.

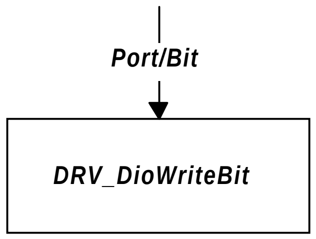


Figure 4-16: Digital Output Function Call Flow

Besides the DRV_DioWriteBit function, the DLL drivers also provide the DRV_DioWritePortByte function to write a byte value to a port. In addition, the DRV_DioGetCurrentDOBit and DRV_DioGetCurrentDOByte functions are used to retrieve current output status.

Examples Directory

\Advantech\Adsapi\Examples\Console\digout

\Advantech\Adsapi\Examples\VB\digout

\Advantech\Adsapi\Examples\Delphi\digout

\Advantech\Adsapi\Examples\VC\digout

Function Description

Demo program for digital output function

4.6 Counter Function Group

The counter function group includes three kinds of operations, event counting, pulse output, and frequency measurement.

4.6.1 Event-Counting

4.6.1.1 General Counter (Intel 8254 or AMD 9513A)

The event-counting functions perform the counter operation.

Call Flow

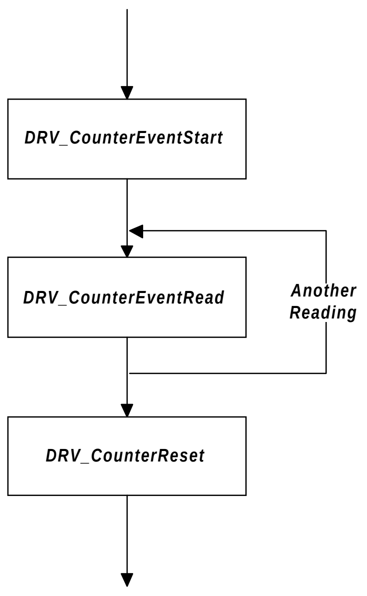


Figure 4-17: General Counter (Intel 8254/AMD 9513A) Function Call Flow

The DRV_CounterEventStart function starts the counter operation. When the counter is running, you can use the DRV_CounterEventRead function to read the count value repeatedly. You can then call the DRV_CounterReset function to stop the counter operation when it is complete.

Notice: a. The programming method depends on the counter/timer chip on the board. There are two kinds of chips that are used in DA&C cards: Intel 8254 and AMD Am9513A. For Am9513A, counter channels 0-9 can all function as a rising edge event counter. Connect your external event generator to the clock input of the desired counter. If hardware "gating", in which the counter may be started by a separate external hardware input, is desired, choose a gating type and use an external device to trigger the gate input of the counter.

b. Both of the above counter/timer chips are 16-bit. However, the function supports a 32-bit counter, i.e., it counts up to 2^{32}. It will check if the counter is overflowing and converts it to 32-bits by calculation.

c. Intel 8254 hardware counter needs 2 cycle times to reload the counter settings, so the counter program has to wait for 2 external triggers (cycle time) to read the correct counter value. At the first time of calling the DRV_CounterEventStart function, Intel 8254 hardware uses a default value to initialize its counter setting. This initialization will take about 2 external triggers (cycle time) to finish. If DRV_CounterEventRead function is called before initialization is finished, then the program will get an incorrect value. You thus have to delay 2 external triggers (cycle time) in the program before calling the DRV_CounterEventRead function to make sure the return value is correct. The delay time is dependent on the time of the external trigger.

Examples Directory

\Advantech\Adsapi\Examples\Console\counter

\Advantech\Adsapi\Examples\VB\counter

\Advantech\Adsapi\Examples\Delphi\counter

\Advantech\Adsapi\Examples\VC\counter

Function Description

Demo program for counter function

4.6.1.2 Interrupt Triggering

The counter operation with interrupt triggering allows you to monitor the background status of the counter operation without polling. When the counter reaches a specified count value, it will acknowledge the driver through a hardware interrupt. These functions are only supported in the PCI-1750/1751.

Call Flow

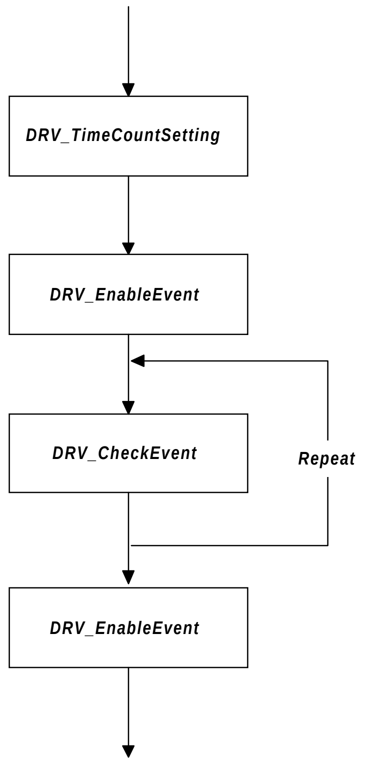


Figure 4-18: Interrupt Triggering Function Call Flow

The DRV_TimerCountSetting function configures the number of interrupt counts. The DRV_EnableEvent function starts the counter operation. When it is running, you can use the DRV_CheckEvent function to check its status. When it reaches the count, it will return a hardware event. You can then use the DRV_EnableEvent function to stop the counter operation.

Examples Directory

\Advantech\Adsapi\Examples\Console\cntint

\Advantech\Adsapi\Examples\VC\cntint

Function Description

Demo program for counter function with interrupt triggering.

4.6.1.3 Quadratic Counting

These functions perform quadratic counter operations. They are only supported by the PCL-833.

Call Flow

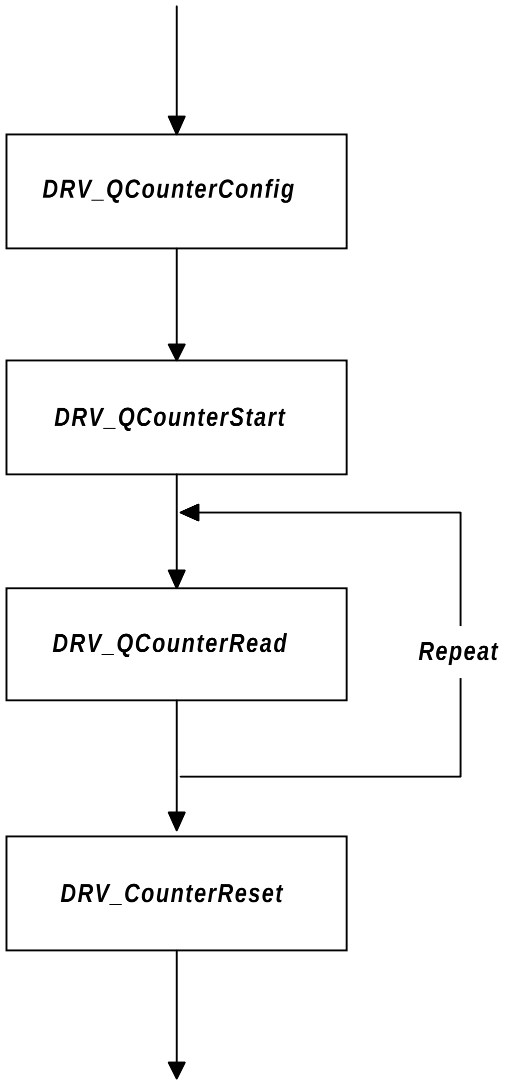


Figure 4-19: Quadratic Counting Function Call Flow

DRV_QCounterConfig function configures the quadratic counter. The DRV_QCounterStart function starts the quadratic counter operation. When it is running, you can use DRV_QCounterRead function to repeatedly read the count value. You can then call DRV_CounterReset to stop the counter operation. In addition, the DLL drivers provide the DRV_QCounterConfigSys function to configure the system clock and time period of the quadratic counter.

Examples Directory

\Advantech\Adsapi\Examples\VC\qcounter

Function Description

Demo program for quadratic counter function

4.6.2 Pulse Output

The pulse output functions include general output and PWM output.

4.6.2.1 General Output

The pulse output functions perform the pulse output operation.

Call Flow

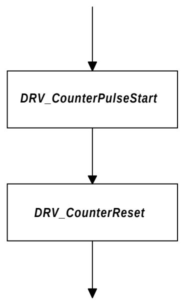


Figure 4-20: Pulse Output (General Output) Function Call Flow

The DRV_CounterPulseStart function starts the pulse output operation. It then runs in the background. When it is complete, you can use the DRV_CounterReset function to stop the pulse output operation.

Notice: *a. The programming method depends on the counter/timer chip on the board. There are two kinds of chips that are used in DA&C cards: Intel 8254 and AMD Am9513A.*

b. For the AMD Am9513A chip, counter channels 0-9 can all function as an arbitrary duty cycle pulse generator. You should select an on-board frequency (F1-F5) source that is closest to the desired output frequency for pulse output. The pulse waveform will then be generated on the output pin of the counter used. If hardware gating, in which the counter may be started by a separate external hardware input, is desired, choose a gating type, and use an external device to trigger the gate input of the counter.

c. The Intel 8254 chip always generates a square wave.

Examples Directory

\Advantech\Adsapi\Examples\Console\pulse

\Advantech\Adsapi\Examples\VB\pulse

\Advantech\Adsapi\Examples\Delphi\pulse

\Advantech\Adsapi\Examples\VC\pulse

Function Description

Demo program for the pulse output function

4.6.2.2 PWM Output

PWM output functions perform the pulse width modulation output.

Call Flow

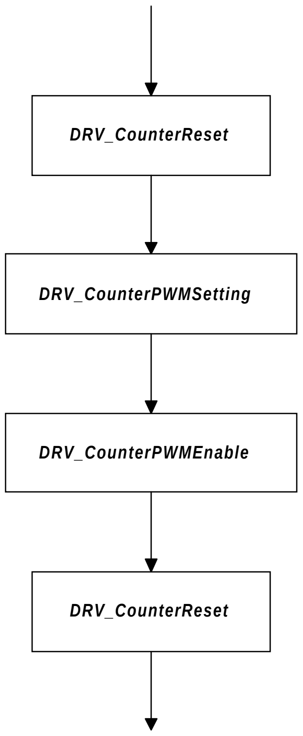


Figure 4-21: PWM Output Function Call Flow

The DRV_CounterReset function resets the PWM counter. The DRV_CounterPWMSetting function configures the PWM counter. The DRV_CounterPWMEnable function enables the PWM pulse output operation. When the operation is complete, you can call the DRV_CounterReset function to stop and reset the PWM pulse output operation.

Examples Directory

\Advantech\Adsapi\Examples\Console\pulsepwm

\Advantech\Adsapi\Examples\VB\pulsepwm

\Advantech\Adsapi\Examples\Delphi\pulsepwm

\Advantech\Adsapi\Examples\VC\pulsepwm

Function Description

Demo program for PWM pulse output function.

4.6.3 Frequency Measurement

The frequency measurement functions measure the pulse frequency.

Call Flow

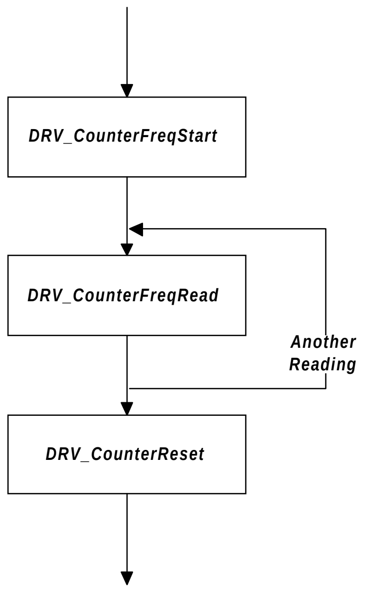


Figure 4-22: Frequency Measurement Function Call Flow

The DRV_CounterFreqStart function starts the frequency measurement. When it is running in the background, you can call the DRV_CounterFreqRead function to repeatedly measure the frequency. You can then call the DRV_CounterReset function to stop the frequency measurement operation.

Notice: *a. The programming method depends on the counter/timer chip on the board. There are two kinds of chips that are used in DA&C cards: Intel 8254 and AMD Am9513A.*

b. Since the AMD Am9513A chip uses two counter/timer channels, a highly accurate frequency measurement device can be attained. Channels 0-8 function as possible input sources for frequency measurement from 1 Hz to 65535 Hz. Channel 9, the last channel on the chip, is reserved and used as a "gate period" counter. For frequency measurement, the on-board time base is used and divided by the "gate period" counter channel. Since a long gating period is generally desirable, choosing F5 (100 Hz) will allow for longer gating periods. You must connect a jumper between the gate period counter output, and the "gate input" of the desired frequency measurement counter. Connect your external frequency generator to the frequency measurement counter's "clock source" input. If hardware "gating", in which the counter may be started by a separate external hardware input, is desired, choose a gating type, and use an external device to trigger the gate input of the gate period counter (fixed at channel 9 by this function).

c. For the Intel 8254 chip, there is no "gate period" counter. The function uses the Windows API to get the time period between two samples. The frequency is then derived from the time period and count increment..

Examples Directory

\Advantech\Adsapi\Examples\Console\freq

\Advantech\Adsapi\Examples\VB\freq

\Advantech\Adsapi\Examples\VC\freq

Function Description

Demo program for frequency measurement function.

4.7 Temperature Measurement Function Group

The temperature measurement function group measures the temperature with expansion boards, such as PCLD-788/789/789D/8115/770.

Parameters

DasChan (Type: Unsigned short, Size: 2 bytes)
It specifies the input channel on the DA&C card.

ExpChan (Type: Unsigned short, Size: 2 bytes)
It specifies the input channel on the expansion board.

DasGain (Type: Unsigned short, Size: 2 bytes)
It specifies the input range or gain on the DA&C card.

TCType (Type: Unsigned short, Size: 2 bytes)
It specifies thermocouple type, J (0), K (1), S (2), T (3), B (4), R (5), and E (6).

TempScale (Type: Unsigned short, Size: 2 bytes)
It specifies the temperature unit, Celsius (0), Fahrenheit (1), Rankie (2), Kelvin (3).

Call Flow

You only have to call DRV_TcMuxRead function to read temperature value

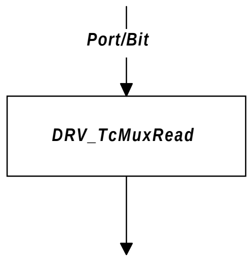


Figure 4-23: Temperature Measurement Function Group Function Call Flow

Expansion Boards

PCLD-770, PCLD-779 or PCLD-789

Follow the procedure to perform thermocouple measurement:

1. Connect the thermocouple(s) to the terminals on the PCLD-770/779/789/889
2. Use a shielded ribbon cable to connect CN1 of the PCLD-770/779/789/889 to the analog input port on the DA&C card in use
3. Use a ribbon cable to connect CN2 of the PCLD-770/779/789 to the digital output port on the DA&C card in use
4. Select a proper input range or gain on the PCLD-770/779/789 for the type of thermocouple used, as described in the PCLD-770/779/789 hardware manual:

```
K type = 50
J type = 100
T type = 200
E type = 50
R type = 200
S type = 200
B type = 200
```

1. Select the desired input channel on the DA&C card to correspond with each PCLD-770/779/789 by setting the jumper block JP1 (PCLD-770), JP16 (PCLD-789) or JP2 (PCLD-779) to a proper position. Positions 0..9 correspond to analog inputs 0..9 of the DA&C card in use.
2. Select the desired input channel on the DA&C card for the CJC (cold junction compensation) circuit on the PCLD-770 by hard wiring the CJC output directly to an A/D channel. On the PCLD-779/789 select the CJC channel by setting the jumper block JP17 (PCLD-789) or JP3 (PCLD-779). Positions 0..9 correspond to analog inputs 0..9 of the DA&C card in use. Of course, the CJC channel selected cannot be set to any analog channel that is already being used for another purpose.
3. If you are cascading or Y-connecting more than one PCLD-779/789 for thermocouple measurement, normally only one CJC input is required - i.e., only one of the PCLD-770/779/789s has to connect its CJC to the DA&C card.
4. Make sure jumper blocks JP16 and JP17 or JP2 and JP3 are not at the same position. They must be set to different input channels on the DA&C card.
5. Select the appropriate configuration in the configuration dialog box of the device installation utility, such as DA&C card, expansion board, expansion gain, and base address, etc.. The driver will perform the appropriate linearization only if the DA&C card's A/D input range is set to –5 V to +5 V.

PCLD-788

Follow the procedure to perform thermocouple measurement:

1. Connect the thermocouple(s) to the PCLD-788 terminals
2. Select the desired input channel on the A/D I/O card to connect to the CJC (cold junction compensation) circuit and connect a jumper from the CJC output to the input channel. Select the same CJC channel during software configuration of the driver. Of course, the CJC channel selected cannot be set to any analog channel being used for another purpose.

3. Configure the base address and connection in the configuration dialog box of the device installation utility.
4. Select the input range -0.05 V to +0.05 V in your application for all thermocouple types.
5. When thermocouple type is selected, the driver will perform the appropriate linearization for the selected thermocouple type with respect to any selected A/D range. However, the optimum range is the A/D range that can handle the entire temperature range for each supported thermocouple type.

PCLD-8115 CJC/Terminal boards

The PCLD-8115 is used as a terminal board to allow the user to connect differential or single-ended signals to a PCL-818HG. The PCLD-8115 includes a CJC circuit that can be enabled or disabled. Because the PCL-818HG provides amplification (to a gain of 1000), the PCLD-8115 itself requires no gain settings. If temperature measurement is to be performed, the CJC (channel 0) must be enabled. The PCLD-8115 must always be connected to the first eight A/D channels (0-7) of the multi-I/O card.

Follow the procedure to perform thermocouple measurement:

1. Connect the thermocouple(s) to the PCLD-8115 terminals.
2. Enable the CJC circuit, and always set at channel 0 on the PCLD-8115. Of course, the CJC channel cannot be used for any other purpose during temperature measurement.
3. Configure the base address and connection in the configuration dialog box of the device installation utility.
4. Select the input range -0.05 V to +0.05 V in your application for all thermocouple types.
5. When thermocouple type is selected, the driver will perform the appropriate linearization for the selected thermocouple type with respect to any selected A/D range. However, the optimum range is the A/D range that can handle the entire temperature range for each supported thermocouple type.

Examples Directory

\Advantech\Adsapi\Examples\Console\thermo

\Advantech\Adsapi\Examples\VB\thermo

\Advantech\Adsapi\Examples\Delphi\thermo

\Advantech\Adsapi\Examples\VC\thermo

Function Description

Demo program for temperature measurement function.

4.8 Alarm Function Group

The alarm functions support Advantech ADAM modules for alarm features. When analog input signal exceeds the range between the pre-defined high or low limits, it will generate an alarm.

Call Flow

The call flow of alarm function is shown below.

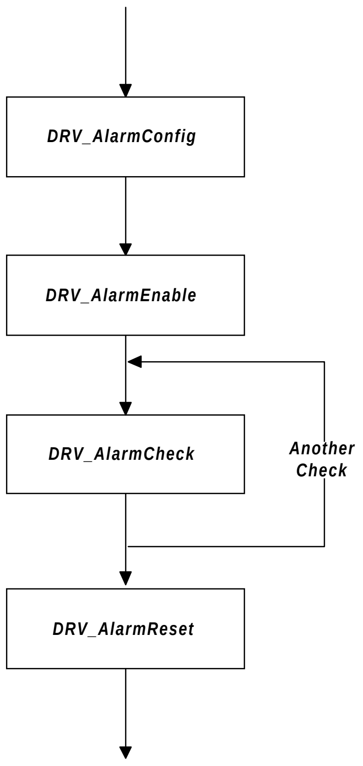


Figure 4-24: Alarm Function Call Flow

The DRV_AlarmConfig function configures the high and low limits for the alarm. The DRV_AlarmEnable function starts the alarm monitoring. While it is running, you can use DRV_AlarmCheck function to check if the alarm occurs repeatedly. When it is complete, you can call DRV_AlarmReset function to stop the alarm operation.

Examples Directory

\Advantech\Adsapi\Examples\VB\alarm

\Advantech\Adsapi\Examples\Delphi\alarm

\Advantech\Adsapi\Examples\VC\alarm

Function Description

Demo program for alarm function.

4.9 Port Function Group

The port function group writes or reads byte/word data to an I/O port. The specified I/O port is an absolute address. The DLL drivers support the following functions:

DRV_outp

Writes byte data from the specified I/O port.

DRV_outpw

Writes word data from the specified I/O port.

DRV_inp

Read byte data from the specified I/O port.

DRV_inpw

Reads word data from the specified I/O port.

Notice: For this function group, you don't need to call DRV_DeviceOpen and DRV_DeviceClose functions.

Examples Directory

\Advantech\Adsapi\Examples\VB\portio

\Advantech\Adsapi\Examples\Delphi\portio

\Advantech\Adsapi\Examples\VC\portio

Function Description

Demo program for port I/O function

4.10 Communication Function Group

The communication function group performs COM port functions.

Call Flow

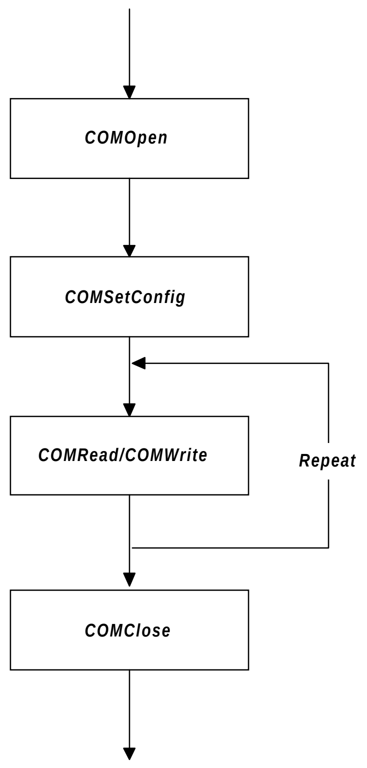


Figure 4-25: Communication Function Call Flow

The COMOpen function opens a serial communication port. This function must be called before using any other functions. The COM-SetConfig function configures the communication port, such as baud rate, parity check, etc.. You can then call COMRead or COMWrite functions repeatedly. When it is complete, you can call COMClose to close the serial communication port.

Other Functions

COMGetConfig

Retrieve the serial port settings; e.g., port number, baud rate, parity check, etc.

COMWrite232

Write data to the specified serial port though the standard RS-232 protocol.

COMEscape

Provide escape services.

Notice: For this function group, you don't need to call DRV_DeviceOpen and DRV_DeviceClose functions. However, you must link adscomm.lib.

Examples Directory

\Advantech\Adsapi\Examples\VC\comm

Function Description

Demo program for communication function

4.11 Event Function Group

Some data acquisition operations run in the background, such as analog input with DMA or interrupt triggering. The DLL drivers provide two ways to check the status of data acquisition operation. One is the polling method. For example, you can call the DRV_FAICheck function repeatedly to check the status of analog input with DMA triggering. The other way is the event function group. After you enable the event function, the DLL driver will fire an event when some hardware interrupt occurs. You do not have to poll the status by yourself.

EnableEvent

First you have to use the EnableEvent function to configure and enable the event type. There are three parameters for it.

Parameters:

Event Type (Data type: Unsigned short, Size: 2 bytes)

The Event Type specifies what kind of event will fire an event. The DLL driver supports the following events.

Type	Value	Description
ADS_EVT_INTERRUPT	0x1	Device generates a hardware interrupt
ADS_EVT_BUFCHANGE	0x2	Buffer changes
ADS_EVT_TERMINATED	0x4	I/O operation is terminated
ADS_EVT_OVERRUN	0x8	Analog input data buffer is overrun
ADS_EVT_WATCHDOG	0x10	Analog input with watchdog triggering is activated
ADS_EVT_PORT0	0x80	Interrupt generated from counter port0
ADS_EVT_PORT1	0x100	Interrupt generated from counter port1
ADS_EVT_PATTERNMATCH	0x200	Pattern matched for digital input
ADS_EVT_COUNTER	0x201	Interrupt for counter
ADS_EVT_COUNTERMATCH	0x202	Count matched for counter
ADS_EVT_COUNTEROVERFLOW	0x203	Counter overflow
ADS_EVT_STATUSCHANGE	0x204	Status change
ADS_EVT_FILTER	0x205	Filter

Table 4-5: Supported Event Types

Enabled (Data type: Unsigned short, Size: 2 bytes)

Set 1 to enable event function or 0 to disable it.

Count (Data type: Unsigned short, Size: 2 bytes)

Specifies how many counts to generate an event.

Notice: ADS_EVT_PORT0 and ADS_EVT_PORT1 are for PCI-1750 and PCI-1751.

CheckEvent

You can use the CheckEvent function to monitor the event status. The CheckEvent function is a synchronous method to check the event. You have to specify a period for the time out. When an event occurs, it returns the event type immediately. If no event occurs in this period, it returns a time out error. The CheckEvent function is different from the DRV_FAICheck or DRV_FAOCheck functions. It uses an efficient polling method to check the event. Your CPU can then simultaneously perform other functions.

EnableEventEx

This function is used in the PCI-1760. It configures and starts the event type for pattern match, digital filter, counter match, counter overflow, or change of state.

Call Flow

Event call flow for single channel analog input with interrupt triggering.

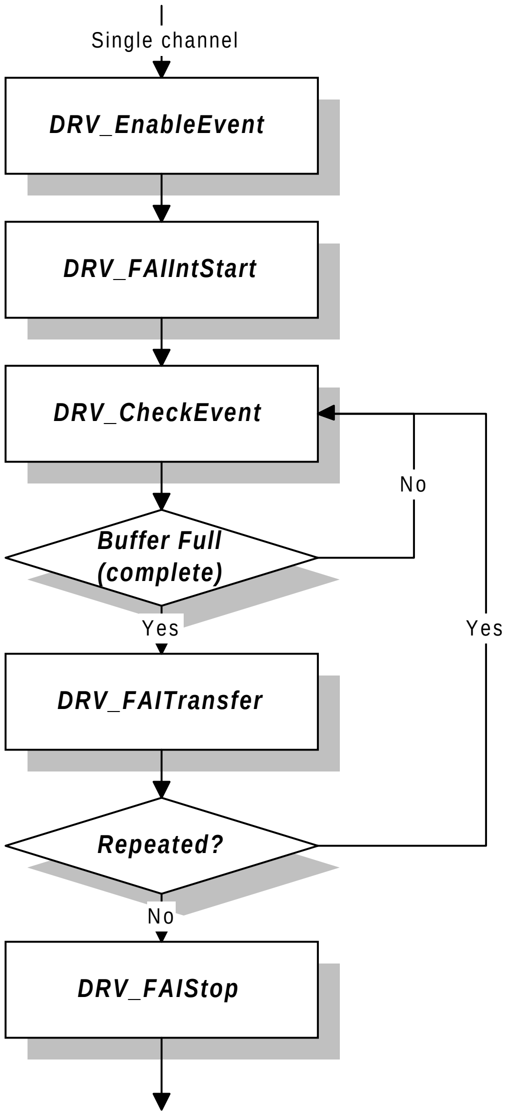


Figure 4-26: Single Channel Analog Input with Interrupt Triggering

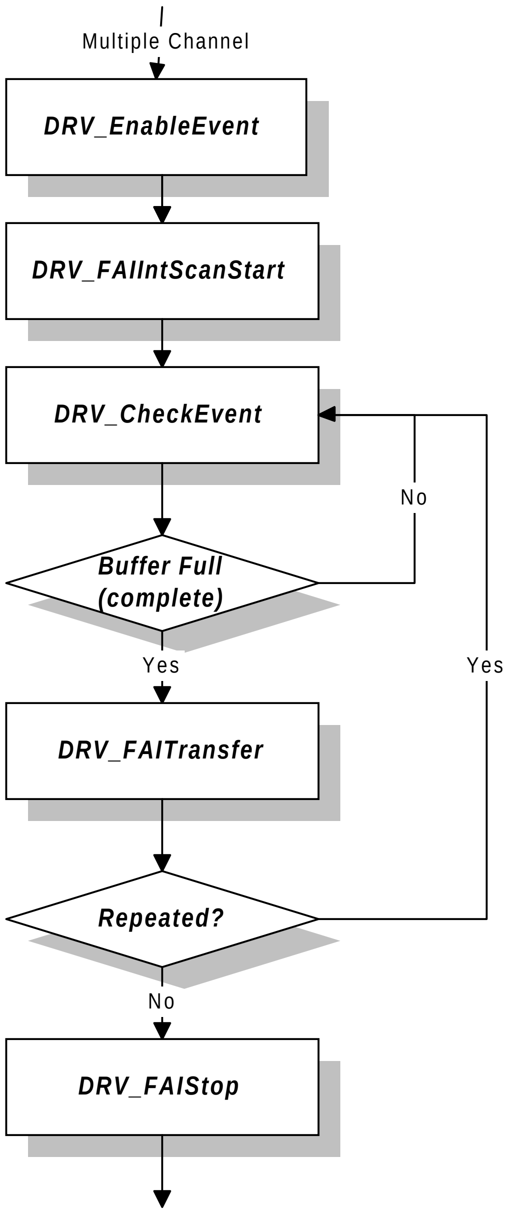


Figure 4-27: Multiple Channel Analog Input with Interrupt Triggering

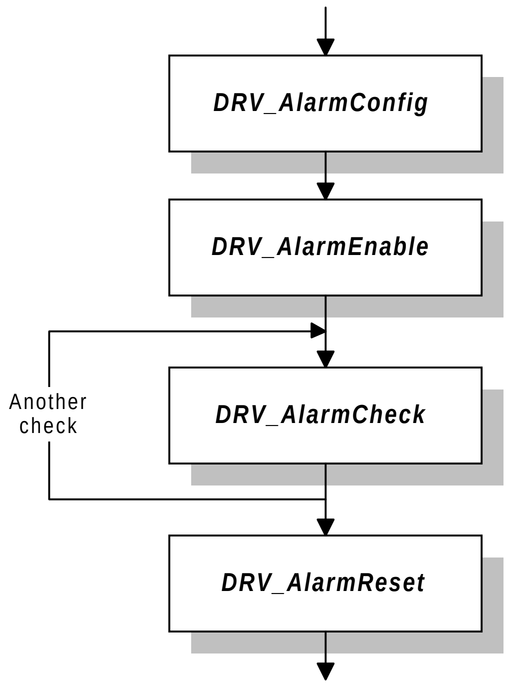


Figure 4-28: Multiple Channel Analog Input with DMA Triggering

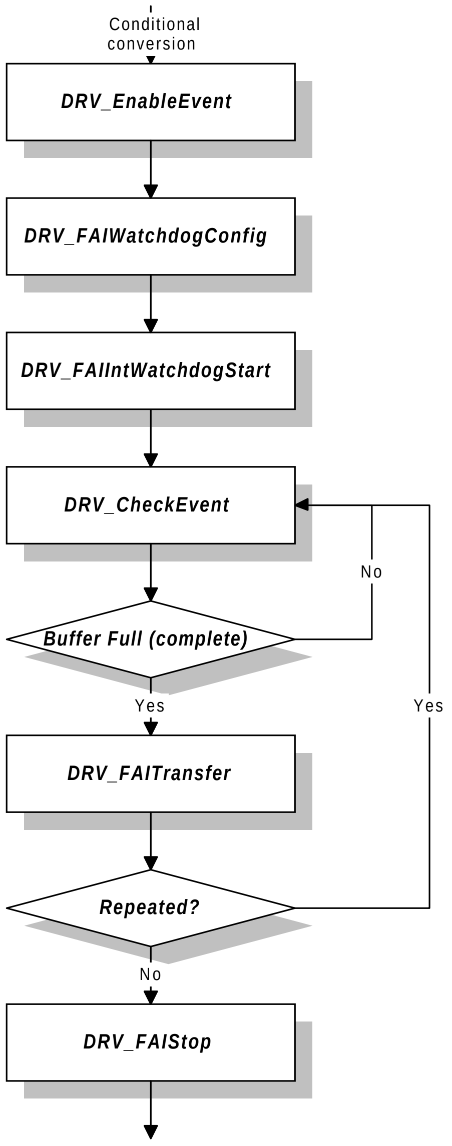


Figure 4-29: Analog Input with Interrupt and Watchdog Triggering

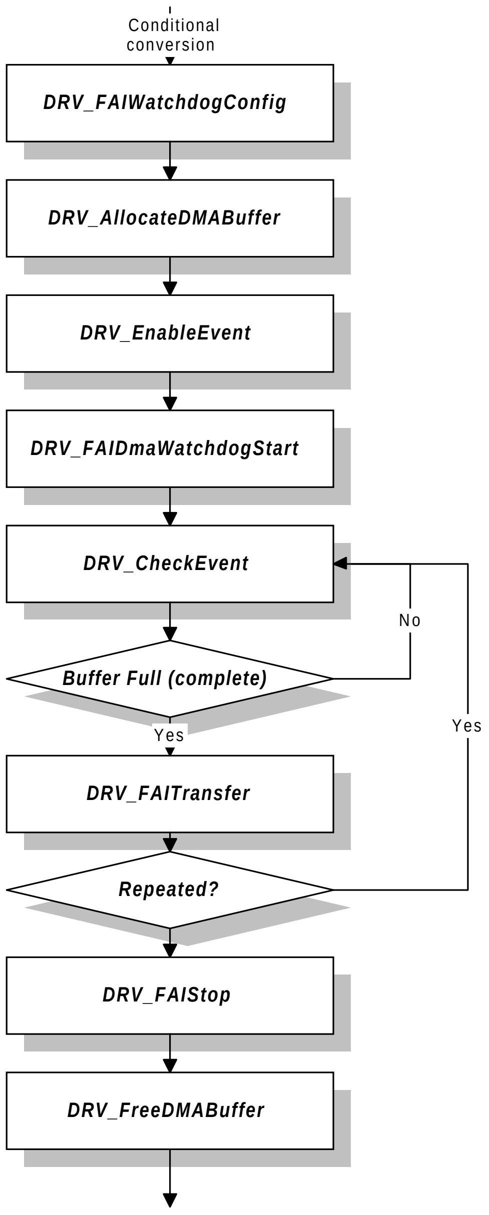


Figure 4-30: Analog Input with DMA and Watchdog Triggering

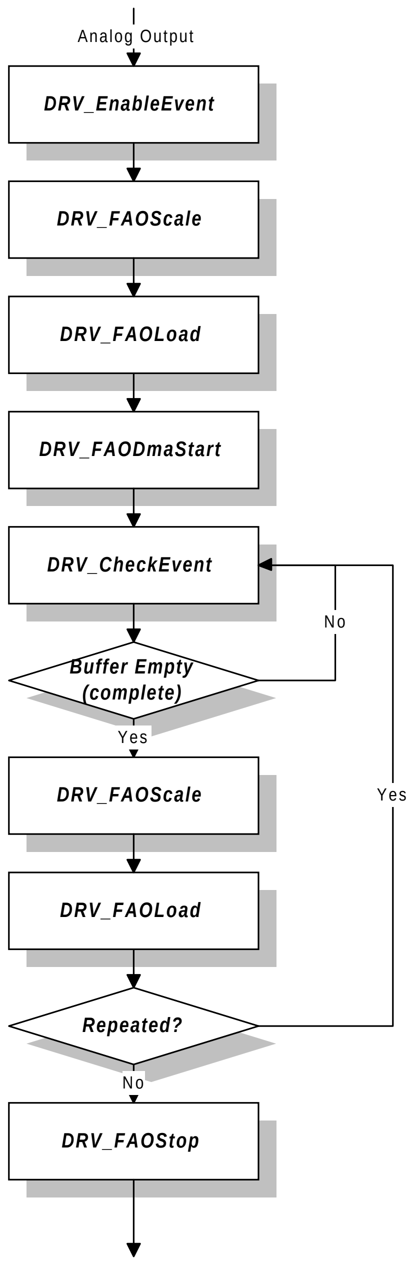


Figure 4-31: Event Call Flow for Analog Output with DMA Triggering

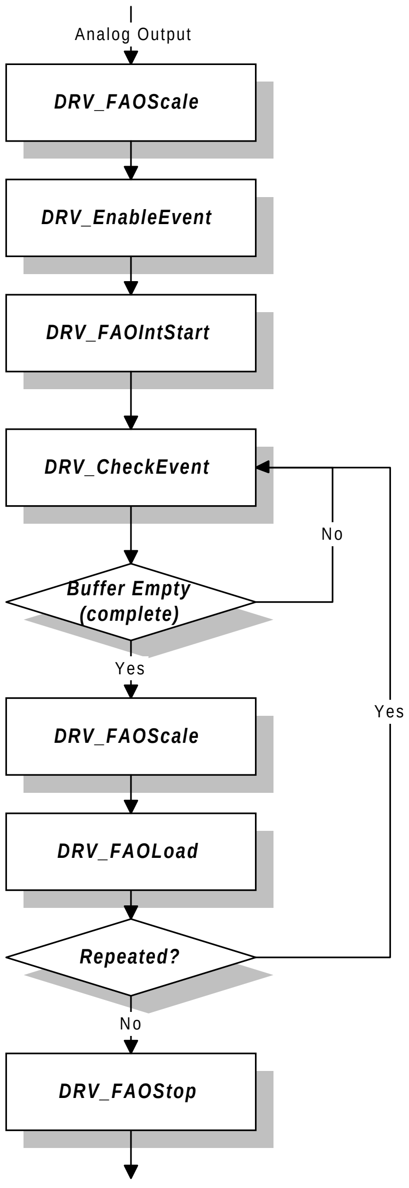


Figure 4-32: Event Call Flow for Analog Output with Interrupt Triggering

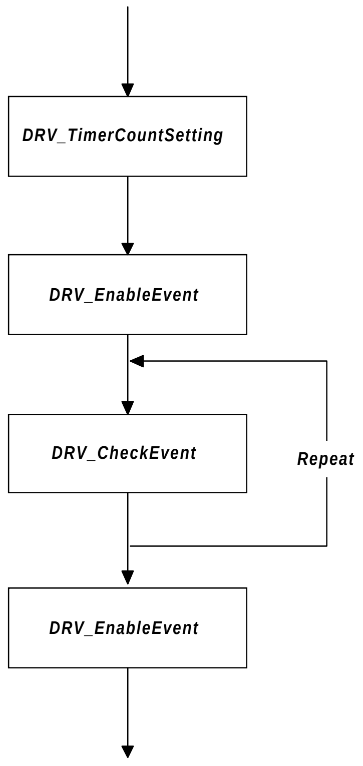


Figure 4-33: Event Call Flow for Counter with Interrupt Triggering

Examples Directory

\Advantech\Adsapi\Examples\VB\addma, adint, cdadint, cdaddma, dipattn

\Advantech\Adsapi\Examples\Delphi\addma, adint, cdadint, cdaddma, dipattn,

\Advantech\Adsapi\Examples\VC\addma, adint, cdadint, cdaddma, diint, dipattn, cntint

Function Description

Demo program for event function

CHAPTER 5

Functions Reference

5.1 Function Support in Advantech Products

No.	File Name	Description
1	ADSAPI32.DLL	Uniform driver for Genie or Application (32-BIT)
2	ADS818.DLL	Support PCL-718/818/818L/818H/818HD/818HG and PCLD-779/788/789D/889
3	ADS711.DLL	Support PCL-711/711B and PCLD-779/788/789D/889
4	ADS812.DLL	Support PCL-812 and PCLD-779/788/789D/889
5	ADS813.DLL	Support PCL-813B/813
6	ADS816.DLL	Support PCL-816/814B and PCLD-779/788/789D/889
7	ADS1800.DLL	Support PCL-1800 and PCLD-779/788/789D/889
8	ADS726.DLL	Support PCL-726/727/728
9	ADSDIO.DLL	Support PCL-720/721/722/723/724/725/730/731/733/734/735
10	ADS4000.DLL	Support ADAM-4011/4012/4013/4014D/4017/4018/4021/4050/4052/4060/4080D/4018M/4530/4521 and PCR-420
11	ADS833.DLL	Support PCL-833
12	ADSMIC.DLL	Support MIC-2730/2732/2750/2752/2718/2728/2760
13	ADSPCM.DLL	Support PCM-3718/3724/3718H/3718HG
14	ADS5000.DLL	Support ADAM-5017/5018/5024/5051/5056/5060 for RS-485 protocol
15	ADSDEMO.DLL	Demo board
16	ADSCOMM.DLL	Support RS-232 function
17	DEVINST.EXE	Device installation utility (32-BIT)
18	DRIVER.H	Function declaration, constant definition for Microsoft C and Borland C
19	DRIVER.BAS	Function declaration, constant definition for Microsoft Visual Basic
20	DRIVER.PAS	Function declaration, constant definition for Borland Delphi
21	ADS836.DLL	Support PCL-836
22	ADS841.DLL	CAN Devices (PCL-841 , MIC-2630, PCM-3680)
23	ADSDN5K.DLL	ADAM-5000 for DeviceNet protocol.
24	ADSIO.DLL	Unlisted Boards for Direct I/O Access
25	ADS1750.DLL	Support PCI-1750
26	ADS1751.DLL	Support PCI-1751
27	ADS1710.DLL	Support PCI-1710
28	ADS1720.DLL	Support PCI-1720
29	ADS1760.DLL	Support PCI-1760
30	ADS1713.DLL	Support PCI-1713
31	ADS1753.DLL	Support PCI-1753

Table 5-1: Driver File Descriptions

5.1.1 Function Support Tables

The following table shows DLL functions that are supported by Advantech hardware.

	Device							
Function	PCL-818 Series	PCL-818HG	PCL-1800	PCL-816	PCL-812PG	PCL-711B	MIC-2718	PCM-3718/H/HG
Device functions								
DRV_DeviceOpen	√	√	√	√	√	√	√	√
DRV_DeviceClose	√	√	√	√	√	√	√	√
DRV_DeviceGetFeatures	√	√	√	√	√	√	√	√
Analog input								
DRV_AIConfig	√	√	√	√	√	√	√	√
DRV_AIGetConfig	√	√	√	√	√	√	√	√
DRV_AIBinaryIn	√	√	√	√	√	√	√	√
DRV_AIScale	√	√	√	√	√	√	√	√
DRV_AIVoltageIn	√	√	√	√	√	√	√	√
DRV_AIVoltageInExp	√	√	√	√	√	√	√	√
DRV_MAIConfig	√	√	√	√	√	√	√	√
DRV_MAIBinaryIn	√	√	√	√	√	√	√	√
DRV_MAIVoltageIn	√	√	√	√	√	√	√	√
DRV_MAIVoltageInExp	√	√	√	√	√	√	√	√
Analog output								
DRV_AOConfig	√	√	√	√	√	√		
DRV_AOBinaryOut	√	√	√	√	√	√		
DRV_AOVoltageOut	√	√	√	√	√	√		
DRV_AOScale	√	√	√	√	√	√		
Port I/O functions								
DRV_WritePortByte	√	√	√	√	√	√	√	√
DRV_WritePortWord	√	√	√	√	√	√	√	√
DRV_ReadPortByte	√	√	√	√	√	√	√	√
DRV_ReadPortWord	√	√	√	√	√	√	√	√
Digital input/output								
DRV_DioGetConfig								
DRV_DioSetPortMode								
DRV_DioReadPortByte	√	√	√	√	√	√		√
DRV_DioWritePortByte	√	√	√	√	√	√		√
DRV_DioReadBit	√	√	√	√	√	√		√
DRV_DioWriteBit	√	√	√	√	√	√		√
DRV_DioGetCurrentDOByte	√	√	√	√	√	√		√
DRV_DioGetCurrentDOBit	√	√	√	√	√	√		√
Temperature								
DRV_TCMuxRead	√	√	√	√	√	√		√

Table 5-2: DLL Functions Supported By Advantech Hardware

	Device							
Function	PCL-818 Series	PCL-818HG	PCL-1800	PCL-816	PCL-812PG	PCL-711B	MIC-2718	PCM-3718/H/HG
High-speed functions								
DRV_FAIIntStart	√	√	√	√	√	√	√	√
DRV_FAIDmaStart	√	√	√	√	√	√		√
DRV_FAIIntScanStart	√	√	√	√	√	√	√	√
DRV_FAIDmaScanStart	√	√	√	√	√	√		√
DRV_FAIDualDmaStart			√					
DRV_FAIDualScanStart			√					
DRV_FAITransfer	√	√	√	√	√	√	√	√
DRV_FAICheck	√	√	√	√	√	√	√	√
DRV_FAIIntWatchdogStart			√					
DRV_FAIDmaWatchdogStart			√					
DRV_FAIWatchdogCheck			√					
DRV_FAIStop	√	√	√	√	√	√	√	√
DRV_AllocateDMABuffer	√	√	√	√	√		√	√
DRV_FreeDMABuffer	√	√	√	√	√		√	√
DRV_FAOIntStart			√	√				
DRV_FAODmaStart			√	√				
DRV_FAOLoad			√	√				
DRV_FAOScale			√	√				
DRV_FAOCheck			√	√				
DRV_FAOStop			√	√				
DRV_CheckEvent	√	√	√	√	√	√	√	√
DRV_CheckEvent	√	√	√	√	√	√	√	√
Counter functions								
DRV_CounterEventStart	√	√	√	√	√	√	√	√
DRV_CounterEventRead	√	√	√	√	√	√	√	√
DRV_CounterFreqStart	√	√	√	√	√	√	√	√
DRV_CounterFreqRead	√	√	√	√	√	√	√	√
DRV_CounterPulseStart	√	√	√	√	√	√	√	√
DRV_CounterReset	√	√	√	√	√	√	√	√
DRV_QCounterConfig								
DRV_QCounterConfigSys								
DRV_QCounterStart								
DRV_QcounterRead								

Table 5-3: DLL Functions Supported By Advantech Hardware

	Device			
Function	PCI-1710	PCI-1713	PCI-1711	PCI-1731
Device functions				
DRV_DeviceOpen	√	√	√	√
DRV_DeviceClose	√	√	√	√
DRV_DeviceGetFeatures	√	√	√	√
Analog input				
DRV_AIConfig	√	√	√	√
DRV_AIGetConfig	√	√	√	√
DRV_AIBinaryIn	√	√	√	√
DRV_AIScale	√	√	√	√
DRV_AIVoltageIn	√	√	√	√
DRV_AIVoltageInExp				
DRV_MAIConfig	√	√	√	√
DRV_MAIBinaryIn	√	√	√	√
DRV_MAIVoltageIn	√	√	√	√
DRV_MAIVoltageInExp				
Analog output				
DRV_AOConfig	√		√	
DRV_AOBinaryOut	√		√	
DRV_AOVoltageOut	√		√	
DRV_AOScale			√	
Port I/O functions				
DRV_WritePortByte	√	√	√	√
DRV_WritePortWord	√	√	√	√
DRV_ReadPortByte	√	√	√	√
DRV_ReadPortWord	√	√	√	√
Digital input/output				
DRV_DioGetConfig				
DRV_DioSetPortMode				
DRV_DioReadPortByte	√		√	√
DRV_DioWritePortByte	√		√	√
DRV_DioReadBit	√		√	√
DRV_DioWriteBit	√		√	√
DRV_DioGetCurrentDOByte	√		√	√
DRV_DioGetCurrentDOBit	√		√	√
Temperature				
DRV_TCMuxRead				

Table 5-4: DLL Driver Functions Supported By PCI-1710/1713/1711/1731

	Device			
Function	PCI-1710	PCI-1713	PCI-1711	PCI-1731
High-speed functions				
DRV_FAIIntStart	√	√	√	√
DRV_FAIDmaStart				
DRV_FAIIntScanStart	√	√	√	√
DRV_FAIDmaScanStart				
DRV_FAIDualDmaStart				
DRV_FAIDualScanStart				
DRV_FAITransfer	√	√	√	√
DRV_FAICheck	√	√		
DRV_FAICheckEvent			√	√
DRV_FAIIntWatchdogStart				
DRV_FAIDmaWatchdogStart				
DRV_FAIWatchdogCheck				
DRV_FAIStop	√	√		
DRV_AllocateDMABuffer				
DRV_FreeDMABuffer				
DRV_FAOIntStart				
DRV_FAODmaStart				
DRV_FAOLoad				
DRV_FAOScale				
DRV_FAOCheck				
DRV_FAOStop				
DRV_CheckEvent	√	√	√	√
DRV_EnableEvent	√	√	√	√
DRV_ClearOverrun	√	√		
Counter functions				
DRV_CounterEventStart	√		√	√
DRV_CounterEventRead	√		√	√
DRV_CounterFreqStart	√		√	√
DRV_CounterFreqRead	√		√	√
DRV_CounterPulseStart	√		√	√
DRV_CounterReset	√		√	√
DRV_QCounterConfig				
DRV_QCounterConfigSys				
DRV_QCounterStart				
DRV_QcounterRead				

Table 5-5: DLL Driver Functions Supported By PCI-1710 /1713/1711/1731

	Device							
Function	PCIA-71A/B/C	PCL-813B	PCL-726/727	PCL-728 MIC2728	Demo Board	PCL-725 /730	PCL-733 MIC-2730 /2732	PCL-722 /724 /731 PCM-3724
Device functions								
DRV_DeviceOpen	√	√	√	√	√	√	√	√
DRV_DeviceClose	√	√	√	√	√	√	√	√
DRV_DeviceGetFeatures	√	√	√	√	√	√	√	√
Analog input								
DRV_AIConfig	√	√			√			
DRV_AIGetConfig	√	√			√			
DRV_AIBinaryIn	√	√						
DRV_AIScale	√	√						
DRV_AIVoltageIn	√	√			√			
DRV_AIVoltageInExp	√	√			√			
DRV_MAIConfig	√	√			√			
DRV_MAIBinaryIn	√	√			√			
DRV_MAIVoltageIn	√	√			√			
DRV_MAIVoltageInExp	√	√			√			
Analog output								
DRV_AOConfig			√	√				
DRV_AOBinaryOut			√	√				
DRV_AOVoltageOut			√	√				
DRV_AOScale			√	√				
Port I/O functions								
DRV_WritePortByte	√	√	√	√	√	√	√	√
DRV_WritePortWord	√	√	√	√	√	√	√	√
DRV_ReadPortByte	√	√	√	√	√	√	√	√
DRV_ReadPortWord	√	√	√	√	√	√	√	√
Digital input/output								
DRV_DioGetConfig								√
DRV_DioSetPortMode								√
DRV_DioReadPortByte	√		√			√	√	√
DRV_DioWritePortByte	√		√			√		√
DRV_DioReadBit	√		√			√	√	√
DRV_DioWriteBit	√		√			√		√
DRV_DioGetCurrentDOByte	√		√			√		√
DRV_DioGetCurrentDOBit	√		√			√		√
Temperature								
DRV_TCMuxRead	√							
High-speed functions								
DRV_CheckEvent								√
DRV_EnableEvent								√

Table 5-6: DLL Driver Functions Supported by Advantech Hardware

	Device
Function	PCI- 1720
Device functions	
DRV_DeviceOpen	√
DRV_DeviceClose	√
DRV_DeviceGetFeatures	√
Analog input	
DRV_AIConfig	
DRV_AIGetConfig	
DRV_AIBinaryIn	
DRV_AIScale	
DRV_AIVoltageIn	
DRV_AIVoltageInExp	
DRV_MAIConfig	
DRV_MAIBinaryIn	
DRV_MAIVoltageIn	
DRV_MAIVoltageInExp	
Analog output	
DRV_AOConfig	√
DRV_AOBinaryOut	√
DRV_AOVoltageOut	√
DRV_AOScale	√
DRV_EnableSyncAO	√
DRV_WriteSyncAO	√
DRV_AOCurrentOut	√
Port I/O functions	
DRV_WritePortByte	√
DRV_WritePortWord	√
DRV_ReadPortByte	√
DRV_ReadPortWord	√
Digital input/output	
DRV_DioGetConfig	
DRV_DioSetPortMode	
DRV_DioReadPortByte	
DRV_DioWritePortByte	
DRV_DioReadBit	
DRV_DioWriteBit	
DRV_DioGetCurrentDOByte	
DRV_DioGetCurrentDOBit	
Temperature	
DRV_TCMuxRead	
High-speed functions	
DRV_CheckEvent	
DRV_EnableEvent	

Table 5-7: DLL Driver Functions Supported by PCI-1720

	Device							
Function	PCL-734/735 MIC-2750/2752/2760	PCL-833	PCL-720	PCL-721/723	PCL-836	PCI-1750	PCI-1751	PCI-1760
Device functions								
DRV_DeviceOpen	√	√	√	√	√	√	√	√
DRV_DeviceClose	√	√	√	√	√	√	√	√
DRV_DeviceGetFeatures	√	√	√	√	√	√	√	√
Port I/O functions								
DRV_WritePortByte	√	√	√	√	√	√	√	√
DRV_WritePortWord	√	√	√	√	√	√	√	√
DRV_ReadPortByte	√	√	√	√	√	√	√	√
DRV_ReadPortWord	√	√	√	√	√	√	√	√
Digital input/output								
DRV_DioGetConfig	√		√	√	√		√	
DRV_DioSetPortMode							√	
DRV_DioReadPortByte		√	√	√	√	√	√	√
DRV_DioWritePortByte	√		√		√	√	√	√
DRV_DioReadBit		√	√	√	√	√	√	√
DRV_DioWriteBit	√		√	√	√	√	√	√
DRV_DioGetCurrentDOByte	√		√	√		√	√	√
DRV_DioGetCurrentDOBit	√		√	√		√	√	√
Counter functions								
DRV_CounterEventStart			√		√	√	√	
DRV_CounterEventRead			√		√	√	√	√
DRV_CounterFreqStart			√		√	√	√	
DRV_CounterFreqRead			√		√	√	√	
DRV_CounterPulseStart			√		√		√	
DRV_CounterReset			√		√	√	√	√
DRV_QCounterConfig		√						
DRV_QCounterConfigSys		√						
DRV_QCounterStart		√						
DRV_QCounterRead		√						
DRV_CounterPWMSetting								√
DRV_CounterPWMEnable								√
DRV_DICounterReset								√
High-speed functions								
DRV_CheckEvent					√	√	√	√
DRV_EnableEvent					√	√	√	
DRV_TimerCountSetting						√	√	
DRV_EnableEventEx								√
DRV_FDITransfer								√

Table 5-8: DLL Driver Function Support by Advantech Hardware

	Device
Function	PCI-1753
Device functions	
DRV_DeviceOpen	√
DRV_DeviceClose	√
DRV_DeviceGetFeatures	√
Port I/O functions	
DRV_WritePortByte	√
DRV_WritePortWord	√
DRV_ReadPortByte	√
DRV_ReadPortWord	√
Digital input/output	
DRV_DioGetConfig	
DRV_DioSetPortMode	√
DRV_DioReadPortByte	√
DRV_DioWritePortByte	√
DRV_DioReadBit	√
DRV_DioWriteBit	√
DRV_DioGetCurrentDOByte	√
DRV_DioGetCurrentDOBit	√
Counter functions	
DRV_CounterEventStart	
DRV_CounterEventRead	
DRV_CounterFreqStart	
DRV_CounterFreqRead	
DRV_CounterPulseStart	
DRV_CounterReset	
DRV_QCounterConfig	
DRV_QCounterConfigSys	
DRV_QCounterStart	
DRV_QCounterRead	
DRV_CounterPWMSetting	
DRV_CounterPWMEnable	
DRV_DICounterReset	
High-speed functions	
DRV_CheckEvent	√
DRV_EnableEvent	√
DRV_TimerCountSetting	
DRV_EnableEventEx	√
DRV_FDITransfer	

Table 5-9: DLL Driver Support for PCI-1753

	Devices							
Function	ADAM-4011/ 4011D	ADAM -4012	ADAM-4014D	ADAM-4018/ 4018M/5018	ADAM-4017/ 4013/5017	ADAM-4021/ 5024	ADAM -4016	ADAM-4052/ 4053/5051/5052
Device functions								
DRV_DeviceOpen	√	√	√	√	√	√	√	√
DRV_DeviceClose	√	√	√	√	√	√	√	√
DRV_DeviceGetFeatu	√	√	√	√	√	√	√	√
Analog input								
DRV_AIGetConfig	√	√	√	√	√		√	
DRV_AIVoltageIn	√	√	√	√	√		√	
DRV_AIVoltageInExp								
DRV_MAIConfig								
DRV_MAIVoltageIn	√	√	√	√	√		√	
DRV_MAIVoltageInEx								
Analog output								
DRV_AOVoltageOut						√		
Digital input/output								
DRV_DioGetConfig								
DRV_DioSetPortMode								
DRV_DioReadPortByt	√	√	√					√
DRV_DioWritePortByt	√	√	√				√	
DRV_DioReadBit	√	√	√					√
DRV_DioWriteBit	√	√	√				√	
DRV_DioGetCurrentD	√	√	√				√	
DRV_DioGetCurrentD	√	√	√				√	
Temperature								
DRV_TCMuxRead	√			√				
Counter functions								
DRV_CounterEventSt	√	√	√					
DRV_CounterEventRe	√	√	√					
DRV_CounterReset	√	√	√					
Alarm functions								
DRV_AlarmConfig	√	√	√	√	√			
DRV_AlarmEnable	√	√	√	√	√			
DRV_AlarmCheck	√	√	√	√	√			
DRV_AlarmReset	√	√	√	√	√			

Table 5-10: DLL Driver Support for Advantech Hardware

Function	ADAM4060/5056/5060	ADAM-4080D	ADAM-4530	ADAM-4521	ADAM-5050	ADAM-4050
Device functions						
DRV_DeviceOpen	√	√	√	√	√	√
DRV_DeviceClose	√	√	√	√	√	√
DRV_DeviceGetFeatures	√	√	√	√	√	√
Digital input/output						
DRV_DioGetConfig					√	
DRV_DioSetPortMode						
DRV_DioReadPortByte					√	√
DRV_DioWritePortByte	√	√			√	√
DRV_DioReadBit					√	√
DRV_DioWriteBit	√	√			√	√
DRV_DioGetCurrentDOByte	√	√			√	√
DRV_DioGetCurrentDOBit	√	√			√	√
Counter functions						
DRV_CounterEventStart		√				
DRV_CounterEventRead		√				
DRV_CounterReset		√				
Alarm functions						
DRV_AlarmConfig		√				
DRV_AlarmEnable		√				
DRV_AlarmCheck		√				
DRV_AlarmReset		√				
Comm. port functions						
COMOpen						
COMClose						
COMGetConfig						
COMSetConfig						
COMRead						
COMWrite						
COMWrite232						
COMWrite485						
COMWrite85						
COMEscape						

Table 5-11: DLL Driver Support for Advantech Hardware

Note: PCL-818HD doesn’t support external trigger function.

5.2 Function Description

The function groups can be analog input, analog output, digital input, digital output... etc. Every Advantech DLL function is of the following form:

```
status = FUNCTION_Name(parameter 1, parameter 2...parameter n)
```

where n^{3}0. Each function returns a value in the status variable that indicates the success or failure of the function as follows:

Status (Value)	Result
UNSUCCESS (> 0)	Function failed due to error
SUCCESS (= 0)	Function completed successfully

Table 5-12: General form of every Advantech DLL Driver Function

Status is a 4-byte integer and is defined in DRIVER.H file. For more information about the error code, please refer to Appendix A.

DRV_SelectDevice

```
status = DRV_SelectDevice(hCaller, GetModule, DeviceNum, Description)
```

Purpose

Showing device list tree dialog box for select device’s number.

Parameters

Name	Direction	Type	Range	Description
hCaller	Input	HWND(windows handle)	Default	Specify the Windows handle that is calling this function.
GetModule	Input	BOOL	TRUE/ FALSE	Specify the function to select module from registry list or not. TRUE (to select module from registry list).
DeviceNum	Output	pointer to unsigned long	0-65535	Return the device number that was get by this function.
Description	Output	pointer to unsigned char	Default	Return the description of the device.

Table 5-13: DRV_SelectDevice Parameter Table

Return:

1. If successful, then return 0
2. If unsuccessful, then return 1

DRV_DeviceGetNumOfList

```
status = DRV_DeviceGetNumOfList(NumOfDevices)
```

Purpose

Gets number of the installed devices.

Parameters

Name	Direction	Type	Range	Description
NumOfDevices	Output	pointer to short	default	returned by the driver

Table 5-14: DRV_DeviceGetNumOfList Parameter Table

Return:

1. SUCCESS if successful
2. InvalidInputParam if NumOfDevice = NULL
3. ConfigDataLost if Registry is missing or invalid

DRV_DeviceGetList

```
status = DRV_DeviceGetList(DeviceList,MaxEntries,OutEntries)
```

Purpose

Gets list of the installed devices not including the attached devices on COM port or CAN.

Parameters

Name	Direction	Type	Range	Description
DeviceList	Output	long pointer to **DEVLIST**		list of the installed devices
MaxEntries	Input	integer	1-999	maximum entries
OutEntries	Output	long pointer to integer	1-999	output entries

Table 5-15: DRV_DeviceGetList Parameter Table

Return

1. SUCCESS if successful
2. InvalidInputParam if NumOfDevice = NULL
3. ConfigDataLost if Registry is missing or invalid

Parameter Details

- **DEVLIST** refers to ADAPI.H.

DRV_DeviceGetSubList

```
status = DRV_DeviceGetSubList(DeviceNum,SubDeviceList,MaxEntries,OutEntries)
```

Purpose

Gets list of the installed devices on COM port or CAN.

Parameters

Name	Direction	Type	Range	Description
DeviceNum	Input	unsigned long	0-999	device number for COM port or CAN
SubDeviceList	Output	long pointer to **DEVLIST**		list of the installed devices
MaxEntries	Input	integer	1-999	maximum entries
OutEntries	Output	long pointer to integer	1-999	output entries

Table 5-16: DRV_DeviceGetSubList Parameter Table

Return

1. SUCCESS if successful
2. InvalidInputParam if NumOfDevice = NULL
3. ConfigDataLost if Registry is missing or invalid

DRV_GetErrorMessage

```
status = DRV_GetErrorMessage(ErrorCode,ErrorMsg)
```

Purpose

Retrieves the message of error according to the error code and returns it in the message buffer.

Parameters

Name	Direction	Type	Range	Description
ErrorCode	Input	unsigned long	default	error code returned by the driver
ErrorMsg	Output	long pointer to string	default	the storage for error message. You have to allocate a minimum of 80 bytes space for the returned error message.

Table 5-17: DRV_GetErrorMessage Parameter Table

Return

1. SUCCESS if successful
2. Fail to Get Error Message if allocate buffer failed
3. Invalid Error Code if error code out of range

Parameter Details

ErrorCode and **ErrorMsg** refer to Appendix: Error Codes.

DRV_DeviceOpen

```
status = DRV_DeviceOpen(DeviceNum,DriverHandle)
```

Purpose

Retrieves parameters pertaining to the device's operation from the Registry or configuration file, and allocate memory to store it for quick reference. This function must be called before any other functions.

Parameters

Name	Direction	Type	Rage	Description
DeviceNum	Input	unsigned long	default	device number
DriverHandle	Output	long pointer	default	a pointer to the configuration data for the device

Table 5-18: DRV_DeviceOpen Parameter Table

Return

1. SUCCESS if successful
2. MemoryAllocateFailed if memory allocation failure
3. ConfigDataLost if retrieving configuration data failure
4. CreateFileFailed if low level driver has an opening failure

Note

1. All subsequent functions perform the desired I/O operations based on configuration data retrieved by the **DriverHandle** parameter.
2. After the I/O operations, user has to call ***DRV_DeviceClose*** to release the memory allocated by ***DRV_DeviceOpen***.

DRV_DeviceClose

```
status = DRV_DeviceClose(DriverHandle)
```

Purpose

Releases the storage allocated by ***DRV_DeviceOpen***.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input/Output	long pointer	default	assigned by ***DRV_DriverOpen***

Table 5-19: DRV_DeviceClose Parameter Table

Return

1. SUCCESS if successful
2. InvalidDriverHandle if DriverHandle = NULL or DriverHandle is not found

DRV_DeviceGetFeatures

```
status = DRV_DeviceGetFeatures(DriverHandle,lpDevFeatures)
```

Purpose

Retrieves the device-specific features and returns them in buffer.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long pointer	default	assigned by ***DRV_DriverOpen***
LpDevFeatures	Output	long pointer to **PT_DEVFEATURES**	default	the storage address of the device features and size of device features

Table 5-20: DRV_DeviceGetFeatures Parameter Table

Parameter Details

- **lpDevFeatures** is the storage address of the device's features and size of device features. The storage layout refers to DRIVER.H.

Return

1. SUCCESS if successful
2. InvalidDriverHandle if DriverHandle = NULL
3. BoardIDNotSupported if Board ID is not supported

See Also

PT_DeviceGetFeatures

DRV_GetAddress

```
Address = DRV_GetAddress (variable)
```

Purpose

This function is only used in Visual Basic. It returns a pointer or address of a variable.

Parameter

Variable: The variable name

Return

A pointer or address of the variable.

DRV_AIConfig

```
status = DRV_AIConfig(DriverHandle,lpAIConfig)
```

Purpose

Configures the gain settings for the specified analog input channel.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAIConfig	Input	long pointer to **PT_AIConfig**	default	the storage address of the DasChan and DasGain

Table 5-21: DRV_AIConfig Parameter Table

Return

- **SUCCESS** if successful
- **InvalidDriverHandle** if DriverHandle = NULL
- **InvalidChannel** if DasChan is out of allowable range
- **InvalidGain** if the DAS card doesn’t support the DasGain
- **BoardIDNotSupported** if Board ID is not supported

DRV_AIGetConfig

```
status = DRV_AIGetConfig(DriverHandle,lpAIGetConfig)
```

Purpose

Retrieves analog input configuration data and returns it in buffer.

Parameters

Name	Direction	Type	Range	Description
DeviceHandle	Input	long pointer	default	assigned by ***DRV_DeviceOpen***
lpAIGetConfig	Input/Output	long pointer to **PT_AIGetConfig**	default	the storage address of the buffer and size

Table 5-22: DRV_AIGetConfig Parameter Table

Return

- **SUCCESS** if successful
- **InvalidDriverHandle** if DriverHandle = NULL

DRV_AIBinaryIn

```
status = DRV_AIBinaryIn(DriverHandle,lpAIBinaryIn)
```

Purpose

Reads an analog input channel and returns the unscaled result.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAIBinaryIn	Input/Output	long pointer to **PT_AIBinaryIn**	default	the storage address for chan, TrigMode and reading

Table 5-23: DRV_AIBinaryIn Function Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **DataNotReady** if the TrigMode=1 and data is not ready
4. **AIConverionFailed** if conversion failed
5. **BoardIDNotSupported** if Board ID is not supported

DRV_AIScale

```
status = DRV_AIScale(DriverHandle, lpAIScale)
```

Purpose

Converts the binary result from an ***DRV_AIBinaryIn*** call or ***DRV_MAIBinaryIn*** call to the actual input voltage.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAIScale	Input/Output	long pointer to **PT_AIScale**	default	the storage address for reading, MaxVolt, MaxCount, offset and Voltage

Table 5-24: DRV_AIScale Function Table

Return

- **SUCCESS** if successful
- **AIScaleFailed** if failure

DRV_AIVoltageIn

```
status = DRV_AIVoltageIn(DriverHandle, lpAIVoltageIn)
```

Purpose

Reads an analog input channel and returns the result scaled to a voltage. (units = volts)

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpVoltageIn	Input/Output	long pointer to **PT_AIVoltageIn**	default	the storage address for chan, gain, TrigMode and voltage

Table 5-25: DRV_AIVoltageIn Function Table

Return

- **SUCCESS** if successful
- **InvalidDriverHandle** if DriverHandle = NULL
- **AIConversionFailed** if A/D conversion failed
- **DataNotReady** if TrigMode=1 and data is not ready
- **AIScale** return code

DRV_AIVoltageInExp

```
status = DRV_AIVoltageInExp(DriverHandle, lpAIVoltageInExp)
```

Purpose

Reads an analog input channel with expansion board and returns the result scaled to a voltage (units = volts). This function supports the expansion boards: PCLD-770/779/789/789D/788.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAIVoltageInExp	Input/Output	long pointer to **PT_AIVoltageInExp**	default	the storage address for DasChan, DasGain, ExpChan and voltage

Table 5-26: DRV_AIVoltageInExp Parameter Table

Return

- **SUCCESS** if successful
- **InvalidDriverHandle** if DriverHandle = NULL
- **AIVoltageIn** return code
- **AIConfig** return code

Note

1. **Configuring the PCL-779, PCLD-770, and PCLD-789/889 Amplifier/Multiplexer Boards**.

The PCLD-789 Amplifier/Multiplexer allows up to 16 differential analog input channels to be multiplexed into one analog input (A/D) channel of a DAS card. The PCLD-779 and PCLD-770 allow for total isolation of up to eight multiplexed channels. When used with multiplexer boards, the first four D/O channels of the DAS card are used for scanning/selecting analog input channels of the PCLD-770, PCLD-779 or PCLD-789 one channel at a time. Because of the digital output allocation when using these expansion boards, the first byte of D/O channels (0-7) on the DAS card are no longer available for standard digital output. Each PCLD-770/779/789/889 must occupy its own A/D channel.

When using mux cards, the input range of the DAS cards should be set for -5 to +5V. You should select the desired gain that is suitable for your signals on the PCLD-779 or PCLD-789.

2. **Cascading PCL-770s, PCLD-779s or PCLD-789s for expansion**

You may use just one PCLD-770, PCLD-779 or PCLD-789 in a stand-alone configuration, or you can cascade up to 8 PCLD-789s (for 128 channels), 8 PCLD-770s (for 64 channels), or Y connect up to four PCLD-779s (for 32 channels, using the optional PCLD-774 Analog Expansion Board), in a system. Jumpers are used on the multiplexed boards to identify the DAS interface card channel to which they will be connected. Each multiplex in the cascade or Y connection must be routed (jumpered) to a different channel on the DAS card. Furthermore, the gain selector switch on each multiplexer should be positioned to select the desired input range.

3. **Configuring the PCLD-788 Relay Multiplexer Boards**

The PCLD-788 Relay Multiplexer allows up to 256 analog input channels to be multiplexed into one analog input (A/D) channel of a DAS card. The first eight D/O channels of the DAS card are used for addressing each PCLD-788 (high nibble) and for scanning/selecting channels (lower nibble) of the PCLD-788 one at a time. Because of the digital output allocation when using these expansion boards, the first byte of D/O channels (0-7) on the DAS card are no longer available for standard digital output. Because of their unique addressing method and very high impedance when not selected, multiple PCLD-788s (up to 16) can be connected to a single analog input (A/D) channel. Therefore, up to 256 channels may be connected to each A/D channel of the I/O card.

When using the PCLD-788, the input range of the DAS card should be set for any desired range.

4. If there is no expansion board connecting to the DAS channel, it calls **DRV_AIVoltageIn**.

DRV_MAIConfig

```
status = DRV_MAIConfig(DriverHandle, lpMAIConfig)
```

Purpose

Configures the gain settings for the specified analog input channels.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpMAIConfig	Input/Output	long pointer to **PT_MAIConfig**	default	the storage address for NumChan, StartChan and GainArray.

Table 5-27: DRV_MAIConfig Parameter Table

Return

- **SUCCESS** if successful
- **InvalidDriverHandle** if DriverHandle = NULL
- **InvalidInputParam** if GainArray = NULL
- **InvalidChannel** if NumChan is out of allowable range
- **AIConfig** return code

DRV_MAIBinaryIn

```
status = DRV_MAIBinaryIn(DriverHandle,lpMAIBinaryIn)
```

Purpose

Reads analog input channels and returns the unscaled results.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpMAIBinaryIn	Input/Output	long pointer to **PT_MAIBinaryIn**	default	the storage address for NumChan, StartChan, TrigMode and ReadingArray.

Table 5-28: DRV_MAIBinaryIn Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidInputParam** if ReadingArray = NULL
4. **AIBinaryIn** return code

DRV_MAIVoltageIn

```
status = DRV_MAIVoltageIn(DriverHandle,lpMAIVoltageIn)
```

Purpose

Reads analog input channels and returns the results scaled to voltages (units = volts)

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpMAIVoltageIn	Input/Output	long pointer to **PT_MAIVoltagein**	default	the storage address for NumChan, StartChan, GainArray, TrigMode and VoltageArray.

Table 5-29: DRV_MAIVoltageIn Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidInputParam** if GainArray or VoltageArray = NULL
4. **AIVoltageIn** return code

DRV_MAIVoltageInExp

```
status = DRV_MAIVoltageInExp(DriverHandle,lpMAIVoltageInExp)
```

Purpose

Reads an analog input channel with expansion board and returns the result scaled to a voltage in units of volts. This function supports the expansion boards: PCLD-770/779/789/789D/788.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpMAIVoltageIn	Input/Output	long pointer to **PT_MAIVoltageIn**	default	the storage address for NumChan, DasChanArray, DasGainArray, ExpChanArray and VoltageArray

Table 5-30: DRV_MAIVoltageInExp Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidInputParam** if DasChanArray, DasGainArray or VoltageArray = NULL
4. **AIVoltageInExp** return code

Note

If there is no expansion boards connecting to the corresponding DAS channel or ExpChanArray is equal to NULL, it will call MAIVoltageIn function. The scan channels must be contiguous, however.

DRV_EnableEventEx

`status =` DRV_EnableEventEx(LONG DriverHandle, LPT_EnableEventEx lpEnableEventEx)

Purpose

Enable or Disable PCI-1760 Event extension. PCI-1760 event extension includes ”Digital Filter”, “Pattern Match”, “Change of Input State”, “Counter Match” and “Counter Overflow”

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	Default	assigned by ***DRV_DeviceOpen***
lpEnableEventEx	Input	long pointer to **PT_EnableEventEx**	Default	the storage address for union structure: ***PT_DIFilter, PT_DIPattern, PT_DIConter and PT_DIStatus***

Table 5-31: DRV_EnableEventEx Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_FDITransfer

`status =` DRV_FDITransfer(LONG DriverHandle, LPT_FDITransfer lpFDITransfer)

Purpose

Access hardware data while event interrupt happened. The event interrupt includes ”Digital Filter”, “Pattern Match”, “Change of Input State”, “Counter Match” and “Counter Overflow”

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFDITransfer	Input/Output	long pointer to **PT_FDITransfer**	default	the storage address for ***usEventTyp*****e** and ***ulRetData***

Table 5-32: DRV_FDITransfer Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_AOConfig

```
status = DRV_AOConfig(DriverHandle,lpAOConfig)
```

Purpose

Records the output range and polarity selected for each analog output channel. It is optional.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAOConfig	Input/Output	long pointer to **PT_AOConfig**	default	the storage address for chan, RefSrc, MaxValue and MinValue

Table 5-33: DRV_AOConfig Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChan** if input channel is out of range
4. **BoardIDNotSupported** if this function is not supported for this Device

Note

1. This function will overwrite the default configuration data.
2. By using this function, it allows the output range to change at run-time.
3. These configuration changes are only temporary at run-time. The configuration data stored in the file or Registry is not modified.

DRV_AOVoltageOut

```
status = DRV_AOVoltageOut(DriverHandle,lpAOVoltageOut)
```

Purpose

Accepts a floating-point voltage value, scales it to the proper binary number, and writes that number to an analog output channel to change the output voltage.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAOVoltageOut	Input/Output	long pointer to **PT_AOVoltageOut**	default	the storage address for chan and OutputValue

Table 5-34: DRV_AOVoltageOut Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChan** if input channel is out of range
4. **BoardIDNotSupported** if this function is not supported for this Device

DRV_AOScale

```
status = DRV_AOScale(DriverHandle,lpAOScale)
```

Purpose

Scales a voltage to a binary value that, when written to one of the analog output channels, produces the specified voltage.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAOScale	Input/Output	long pointer to **PT_AOScale**	default	the storage address for chan, OutputValue and BinData

Table 5-35: DRV_AOScale Parameter Table

Return :

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if this function is not supported for this Device

DRV_AOBinaryOut

```
status = DRV_AOBinaryOut(DriverHandle,lpAOBinaryOut)
```

Purpose

Writes a binary value to one of the analog output channels, changing the voltage produced at the channel.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAOBinaryOut	Input/Output	long pointer to **PT_AOBinaryOut**	default	the storage address for chan and BinData

Table 5-36: DRV_AOBinaryOut Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChan** if input channel is out of range
4. **BoardIDNotSupported** if this function is not supported for this device

DRV_EnableSyncAO

`status =` DRV_EnableSyncAO(DriverHandle, Enable)

Purpose

Enable or Disable synchronized output operation. Note that synchronized output should be configured using DRV_WriteSyncAO before it is enabled or disabled.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	Default	assigned by ***DRV_DeviceOpen***
Enable	Input	unsign short	Default	Enable channel data

Table 5-37: DRV_EnableSyncAO Parameter Table

Return :

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChan** if input channel is out of range
4. **BoardIDNotSupported** if this function is not supported for this device

DRV_WriteSyncAO

`status =` DRV_WriteSyncAO(DriverHandle)

Purpose

Set the Synchronized output bit for synchronized output operation

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	Default	assigned by ***DRV_DeviceOpen***

Table 5-38: DRV_WriteSyncAO Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_AOCurrentOut

`status =` DRV_AOCurrentOut(DriverHandle, lpAOCurrentOut)

Purpose

Output value to Current Sink Connections

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpAOCurrentOut	Input/Output	long pointer to **AOCurrentOut**	default	the storage address for ***usEventType*** and ***OutputValue***

Table 5-39: DRV_AOCurrentOut Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_DioGetConfig

```
status = DRV_DioGetConfig(DriverHandle,lpDioGetConfig)
```

Purpose

Retrieves digital input/output configuration data and returns it in PortArray.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioGetConfig	Input/Output	long pointer to **PT_DioGetConfig**	default	the storage address for ***PortArray*** and ***NumOfPorts***

Table 5-40: DRV_DioGetConfig Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_DioSetPortMode

```
status = DRV_DioSetPortMode(DriverHandle,lpDioSetPortMode)
```

Purpose

Configures the specified port for input or output. This function only supports PCL-722/724/731.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioSetPortMode	Input/Output	long pointer to **PT_DioSetPort Mode**	default	the storage address for port and dir

Table 5-41: DRV_DioSetPortMode Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if this function is not supported for this device

Note

1. By using this function, it allows the DIO port for input or output to change at run-time.

DRV_DioReadPortByte

```
status = DRV_DioReadPortByte(DriverHandle,lpDioReadPortByte)
```

Purpose

Returns digital input data from the specified digital I/O port. The byte is specified by port number which is from 0 to the maximum byte of the device's digital output. For example, PCL-722 has up to 18 digital output ports. The port number of the board is from 0 to 17.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpDioReadPortByte	Input/Output	long pointer to **PT_DioReadPortByte**	default	the storage address for port and value

Table 5-42: DRV_DioReadPortByte Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

DRV_DioWritePortByte

```
status = DRV_DioWritePortByte(DriverHandle,lpDioWritePortByte)
```

Purpose

Writes digital output data to the specified digital port.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioWritePortByte	Input/Output	long pointer to **PT_DioWritePortByte**	default	the storage address for port, mask and state

Table 5-43: DRV_DioWritePortByte Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

Note

The previous state of the digital port should be stored with the configuration data.

DRV_DioReadBit

```
status = DRV_DioReadBit(DriverHandle,lpDioReadBit)
```

Purpose

Returns the bit state of digital input from the specified digital I/O port. The byte is specified by port number which is from 0 to the maximum byte of the device's digital output. For example, PCL-722 has up to 18 ports digital output. The port number of the board is from 0 to 17.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioReadBit	Input/Output	long pointer to **PT_DioReadBit**	default	the storage address for port, bit and state

Table 5-44: DRV_DioReadBit Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range
5. **InvalidInputParam** if bit number is greater than 7

DRV_DioWriteBit

```
status = DRV_DioWriteBit(DriverHandle,lpDioWriteBit)
```

Purpose
Writes digital output data to the specified digital port. The byte is specified by the port number which is from 0 to the maximum byte of the device's digital output. For example, PCL-730 has 4 bytes digital output. The port number of the board is from 0 to 3.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioWriteBit	Input/Output	long pointer to **PT_DioWriteBit**	default	the storage address for port, bit and state

Table 5-45: DRV_DioWriteBit Parameter Table

Return
1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range
5. **InvalidInputParam** if bit number is greater than 7

Note
The previous state of the digital port should be stored with the configuration data.

DRV_DioGetCurrentDOByte

```
status = DRV_DioGetCurrentDOByte(DriverHandle,lpDioGetCurrentDOByte)
```

Purpose

Returns digital input data from the specified digital I/O port. The byte is specified by port number which is from 0 to the maximum byte of the device's digital output. For example, PCL-722 has up to 18 ports digital output. The port number of the board is from 0 to 17.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioGetCurrentDOByte	Input/Output	long pointer to **PT_DioGetCurrentDOByte**	default	the storage address for port and value

Table 5-46: DRV_DioGetCurrentDOByte

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

DRV_DioGetCurrentDOBit

```
status = DRV_DioGetCurrentDOBit(DriverHandle,lpDioGetCurrentDOBit)
```

Purpose

Returns the bit data of digital input from the specified digital I/O port. The byte is specified by port number which is from 0 to the maximum byte of the device's digital output. For example, PCL-722 has up to 18 ports digital output. The port number of the board is from 0 to 17.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpDioGetCurrentDOBit	Input/Output	long pointer to **PT_DioGetCurrentDOBit**	default	the storage address for port, bit and state

Table 5-47: DRV_DioGetCurrentDOBit Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range
5. **InvalidInputParam** if bit number is greater than 7

DRV_WritePortByte

```
status = DRV_WritePortByte(DriverHandle,lpWritePortByte)
```

Purpose

Writes an 8-bit data to the specified I/O port. The port address is an I/O port address on the PC.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpWritePortByte	Input/Output	long pointer to **PT_WritePortByte**	default	the storage address for port and ByteData

Table 5-48: DRV_WritePortByte Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **KeInvalidHandleValue** if the kernel mode driver cannot be opened
4. **KeFileNotFound** if an attempt was made to open kernel mode driver while the driver was not running.
5. **KeTooManyCmds** if the logic commands have created an apparent endless loop for kernel mode driver.
6. **KeInvalidHandle** if the handle for kernel mode driver is not a valid handle.
7. **KeInvalidParameter** if the parameter passed to kernel mode driver is incorrect.
8. **KeNoAccess** if an attempt to access a port or memory address which has not been defined in the Registry for this device.

DRV_WritePortWord

```
status = DRV_WritePortByte(DriverHandle,lpWritePortWord)
```

Purpose

Writes a 16-bit data to the specified I/O port. The port address is an I/O port address on the PC.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpWritePortWord	Input/Output	long pointer to PT_WritePortWord	default	the storage address for port and WordData

Table 5-49: DRV_WritePortWord Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **KeInvalidHandleValue** if the kernel mode driver cannot be opened
4. **KeFileNotFound** if an attempt was made to open kernel mode driver while the driver was not running.
5. **KeTooManyCmds** if the logic commands have created an apparent endless loop for kernel mode driver.
6. **KeInvalidHandle** if the handle for kernel mode driver is not a valid handle.
7. **KeInvalidParameter** if the parameter passed to kernel mode driver is incorrect.
8. **KeNoAccess** if an attempt to access a port or memory address which has not been defined in the Registry for this device.

DRV_ReadPortByte

```
status = DRV_ReadPortByte(DriverHandle,lpReadPortByte)
```

Purpose

Reads an 8-bit data from the specified I/O port. The port address is an I/O port address on the PC.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpReadPortByte	Input/Output	long pointer to **PT_ReadPortByte**	default	the storage address for port and ByteData

Table 5-50: DRV_ReadPortByte Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **KeInvalidHandleValue** if the kernel mode driver cannot be opened
4. **KeFileNotFound** if an attempt was made to open kernel mode driver while the driver was not running.
5. **KeTooManyCmds** if the logic commands have created an apparent endless loop for kernel mode driver.
6. **KeInvalidHandle** if the handle for kernel mode driver is not a valid handle.
7. **KeInvalidParameter** if the parameter passed to kernel mode driver is incorrect.
8. **KeNoAccess** if an attempt to access a port or memory address which has not been defined in the Registry for this device.

DRV_ReadPortWord

```
status = DRV_ReadPortWord(DriverHandle,lpReadPortWord)
```

Purpose

Reads a 16-bit data from the specified I/O port. The port address is an I/O port address on the PC.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpReadPortWord	Input/Output	long pointer to **PT_ReadPortWord**	default	the storage address for port and WordData

Table 5-51: DRV_ReadPortWord Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **KeInvalidHandleValue** if the kernel mode driver cannot be opened
4. **KeFileNotFound** if an attempt was made to open kernel mode driver while the driver was not running.
5. **KeTooManyCmds** if the logic commands have created an apparent endless loop for kernel mode driver.
6. **KeInvalidHandle** if the handle for kernel mode driver is not a valid handle.
7. **KeInvalidParameter** if the parameter passed to kernel mode driver is incorrect.
8. **KeNoAccess** if an attempt to access a port or memory address which has not been defined in the Registry for this device.

DRV_CounterEventStart

```
status =
DRV_CounterEventStart(DriverHandle,lpCounterEventStart)
```

Purpose

Configures the specified counter for an event-counting operation and starts the counter.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpCounterEventStart	Input/Output	long pointer to **PT_CounterEventStart**	default	the storage address for countger and GateMode

Table 5-52: DRV_CounterEventStart Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

Operations

1. The programming method depends on the counter/timer chip on the board. There are two kinds of chips that are used in DASCards: Intel 8254 and AMD Am9513A. For Am9513A, counter channels 0-9 can all function as a rising edge event counter. Connect your external event generator to the clock input of the desired counter. If hardware “gating”, in which the counter may be started by a separate external hardware input, is desired, choose a gating type and use an external device to trigger the gate input of the counter.
2. Both of the above counter/timer chips are 16-bits. However, the function supports a 32-bit counter, i.e. it counts up 2^{32}. It will check if the counter is overflowing and converts it to 32-bits by calculation.

3. Intel 8254 hardware counter needs 2 cycle time to reload counter setting, so counter program has to wait for 2 external trigger (cycle time) to read correct counter value. At the first time of calling "DRV_CounterEventStart", Intel 8254 hardware uses default value to initialize its counter setting. This initialization will take about 2 external trigger (cycle time) to finish. If "DRV_CounterEventRead" is called before initialization is finished, then the program will get incorrect value. So, you have to delay 2 external trigger (cycle time) in program before calling "DRV_CounterEventRead" to make sure the return value is correct. The delay time is dependent of the time of external trigger.

DRV_CounterEventRead

```
status = DRV_CounterEventRead(DriverHandle,lpCounterEventRead)
```

Purpose

Reads the current counter total without disturbing the counting process and returns the count and overflow conditions.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpCounterEventRead	Input/Output	long pointer to **PT_CounterEventRead**	default	the storage address for counter, overflow and count

Table 5-53: DRV_CounterEventRead Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

DRV_CounterFreqStart

```
status = DRV_CounterFreqStart(DriverHandle,lpCounterFreqStart)
```

Purpose

Configures the specified counter for frequency measurement and starts the counter.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpCounterFreqStart	Input/Output	long pointer to **PT_CounterFreqStart**	default	the storage address for counter, GatePeriod and GateMode

Table 5-54: DRV_CounterFreqStart Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

Operations

1. The programming method depends on the counter/timer chip on the board. There are two kinds of chips that are used in DASCards: Intel 8254 and AMD Am9513A.

Since the AMD Am9513A chip uses two counter/timer channels, a highly accurate frequency measurement device can be attained. Channels 0-8 function as possible input sources for frequency measurement from 1 Hz to 65535 Hz. Channel 9, the last channel on the chip, is reserved and used as a “gate period” counter. For frequency measurement, the on-board time base is used and divided by the “gate period” counter channel. Since a long gating period is generally desirable, choosing F5 (100 Hz) will allow for longer gating periods. You must connect a jumper between the gate period counter output, and the “gate input” of the desired frequency measurement counter.

Connect your external frequency generator to the frequency measurement counter's "clock source" input. If hardware "gating", in which the counter may be started by a separate external hardware input, is desired, choose a gating type, and use an external device to trigger the gate input of the gate period counter (fixed at channel 9 by this function).

For Intel 8254 chip, there is no "gate period" counter. The function uses the Windows API to get the time period between two samples. The frequency is then derived from the time period and count increment.

DRV_CounterFreqRead

```
status = DRV_CounterFreqRead(DriverHandle,lpCounterFreqRead)
```

Purpose

Reads the frequency measurement.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpCounterFreqRead	Input/Output	long pointer to **PT_CounterFreqRead**	default	the storage address for counter and freq

Table 5-55: DRV_CounterFreqRead Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range
5. **FreqMeasurementFailed** if the time interval for frequency measurement is too small

DRV_CounterPulseStart

```
status = DRV_CounterPulseStart(DriverHandle,lpCounterPulseStart)
```

Purpose

Configures the specified counter for pulse output and starts the counter.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpCounterPulseStart	Input/ Output	long pointer to **PT_CounterPulseStart**	default	the storage address for counter, period, UpCycle and GateMode

Table 5-56: DRV_CounterPulseStart Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

Operations

1. The programming method depends on the counter/timer chip on the board. There are two kinds of chips that are used in DASCards: Intel 8254 and AMD Am9513A.
2. For the AMD Am9513A chip, counter channels 0-9 can all function as an arbitrary duty cycle pulse generator. You should select an on-board frequency (F1-F5) source that is closest to the desired output frequency for pulse output. The pulse waveform will then be generated on the output pin of the counter used. If hardware gating, in which the counter may be started by a separate external hardware input, is desired, choose a gating type, and use an external device to trigger the gate input of the counter.

The Intel 8254 chip always generates a square wave. Hence it does not use the **UpCycle**.

DRV_CounterReset

```
status = DRV_CounterReset(DriverHandle,counter)
```

Purpose

Turns off the specified counter operation. This function supports boards with the timer/counter chip (i.e. Intel 8254 or AMD Am9513A) and PCL-833.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
counter	Input	long	default	counter channel

Table 5-57: DRV_CounterReset Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

DRV_QCounterConfig

```
status = DRV_QCounterConfig(DriverHandle,lpQCounterConfig)
```

Purpose

Configures the specified counter for an event-counting operation. This function only supports PCL-833.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpQCounterConfig	Input/Output	long pointer to **PT_QCounterConfig**	default	the storage address for counter, LatchSrc, LatchOverflow, ResetOnLatch and ResetValue

Table 5-58: DRV_QcounterConfig Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

DRV_QCounterConfigSys

```
status = DRV_QCounterConfigSys(DriverHandle,lpQCounterConfigSys)
```

Purpose

Configures system clock of the digital filter, time period for latching and cascade mode. This function only supports PCL-833.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpQCounterConfigSys	Input/Output	long pointer to **PT_QCounterConfnigSys**	default	the storage address for SysClock, LatchPeriod and CascadeMode

Table 5-59: DRV_QcounterConfigSys Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device

DRV_QCounterStart

```
status = DRV_QCounterStart(DriverHandle,lpQCounterStart)
```

Purpose

Configures the specified counter for an event-counting operation and starts the counter. This function only supports PCL-833.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpQCounterStart	Input/Output	long pointer to **PT_QCounterStart**	default	the storage address for counter and InputMode

Table 5-60: DRV_QcounterStart Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

DRV_QCounterRead

```
status = DRV_QCounterRead(DriverHandle,lpQCounterRead)
```

Purpose

Reads the current counter total without disturbing the counting process and returns the count and overflow conditions.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpQCounterRead	Input/Output	long pointer to **PT_QCounterRead**	default	the storage address for counter, overflow, LoCount and HiCount

Table 5-61: DRV_QcounterRead Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

Note

When not in cascade mode, the counter is 24-bits. The data only returns in **LoCount**. Otherwise, when it is in cascade mode, the counter is 48-bits. The data returns in **LoCount** and **HiCount**.

DRV_DICounterReset

`status =` DRV_DICounterReset(DriverHandle, counter)

Purpose

Reset the value of specified counter to be reset value

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
counter	Input	unsign short	default	reset counter data

Table 5-62: DRV_DICounterReset Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

Note

DRV_CounterPWMSetting

`status =` DRV_CounterPWMSetting(DriverHandle, lpCounterPWMSetting)

Purpose

Config the setting value of PWM(Pulse Width Modulation) output

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpCounterPWMSetting	Input	long pointer to **PT_CounterPWMSetting**	default	the storage address for ***Port, Period, HiPeriod, OutCount*** and ***GateMode***

Table 5-63: DRV_CounterPWMSetting Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

Note

DRV_CounterPWMEnable

`status =` DRV_CounterPWMEnable(DriverHandle, Port)

Purpose

Enable PWM(Pulse Width Modulation) output operation

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
Port	Input	unsign Short	default	Enable/Disable port. If bit0 = 1, port0 is enabled. If bit1 = 1, port1 is enabled.

Table 5-64: DRV_CounterPWMEnable Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

Note

DRV_TCMuxRead

```
status = DRV_TCMuxRead(DriverHandle,lpTCMuxRead)
```

Purpose

Measures the temperature with expansion boards, for example, PCLD-788/779/789D/8115/770.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpTCMuxRead	Input/Output	long pointer to **PT_TCMuxRead**	default	the storage address for DasChan, DasGain, ExpChan, TCType, TempScale and temp

Table 5-65: DRV_TCMuxRead Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **NoExpansionBoardConfig** if no expansion board is attached
4. **TCExp8115Read** return code
5. **TCExp788Read** return code

Note:
Using the PCLD-770, PCLD-779 or PCLD-789 for thermocouple measurement. Thermocouple linearization is provided automatically by the driver if a temperature measurement operation is chosen in the application program. The linearization is performed, and the temperature acquired by the thermocouple/mux card is available for control strategy use or display in degrees Celsius. The conversion to units other than degrees C (Fahrenheit, Kelvin, etc.) can be accomplished by use of a calculation scaling factor.

To perform thermocouple measurement:

1. Properly configure the DAS card
2. Connect the thermocouple(s) to the terminals on the PCLD-770/779/789/889
3. Use a shielded ribbon cable to connect CN1 of the PCLD-770/779/789/889 to the analog input port on the DAS card in use
4. Use a ribbon cable to connect CN2 of the PCLD-770/779/789 to the digital output port on the DAS card in use
5. Select a proper input range or gain on the PCLD-770/779/789 for the type of thermocouple used, as described in the PCLD-770/779/789 hardware manual:

```
K type = 50
J type = 100
T type = 200
E type = 50
R type = 200
S type = 200
B type = 200
```

6. Select the desired input channel on the DAS card to correspond with each PCLD-770/779/789 by setting jumper block JP1 (PCLD-770), JP16 (PCLD-789) or JP2 (PCLD-779) to a proper position. Positions 0..9 correspond to analog inputs 0..9 of the DAS card in use.

7. Select the desired input channel on the DAS card for the CJC (cold junction compensation) circuit on the PCLD-770 by hard wiring the CJC output directly to an A/D channel. On the PCLD-779/789 select the CJC channel by setting the jumper block JP17 (PCLD-789) or JP3 (PCLD-779). Positions 0..9 correspond to analog inputs 0..9 of the DAS card in use. Of course, the CJC channel selected cannot be set to any analog channel that is already being used for another purpose.
8. If you are cascading or Y-connecting more than one PCLD-779/789 for thermocouple measurement, normally only one CJC input is required - i.e. only one of the PCLD-770/779/789s has to connect its CJC to the DAS card.
9. Make sure jumper blocks JP16 and JP17 or JP2 and JP3 are not at the same position. They must be set to different input channels on the DAS card.
10. Select the appropriate configuration in the driver configuration dialog box - DAS card, expansion board, and base address, etc.
11. When THERMOCOUPLE TYPE in the application software is selected, the driver will perform the appropriate linearization only if the DAS card's A/D input range is set to -5V to +5V.

Using the PCLD-788 for thermocouple measurement

Thermocouple linearization is provided by the driver automatically if a temperature measurement operation is chosen in the application program. The linearization is performed, and the temperature acquired by the thermocouple/mux card is available for control strategy use or display in Degrees Celsius. The conversion to units other than degrees C (Fahrenheit, Kelvin, etc.) can be accomplished by using a calculation scaling factor. To perform thermocouple measurement:

1. Properly configure the DAS card
2. Connect the thermocouple(s) to the PCLD-788 terminals
3. Select the desired input channel on the A/D I/O card to connect to the CJC (cold junction compensation) circuit and connect a jumper from the CJC output to the input channel. Select the same CJC channel during software configuration of the driver. Of course, the CJC channel selected cannot be set tot any analog channel being used for another purpose.

4. Select the appropriate configuration in the driver configuration dialog box - base address, etc..
5. Select the input range -0.05V to +0.05V in the application software for all thermocouple types
6. When THERMOCOUPLE TYPE in the application software is selected, the driver will perform the appropriate linearization for the selected thermocouple type with respect to any selected A/D range. However, the optimum range is the A/D range that can handle the entire temperature range for each supported thermocouple type.

Configuring the PCLD-8115 CJC/Terminal boards

The PCLD-8115 is used as a terminal board to allow the user to connect differential or single-ended signals to a PCL-818HG. The PCLD-8115 includes a CJC circuit that can be enabled or disabled. Because the PCL-818HG provides amplification (to a gain of 1000), the PCLD-8115 itself requires no gain settings. If temperature measurement is to be performed, the CJC (channel 0) must be enabled. The PCLD-8115 must always be connected to the first eight A/D channels (0-7) of the multi-I/O card.

Using the PCLD-8115 for thermocouple measurement

Thermocouple linearization is provided by the driver automatically if a temperature measurement operation is chosen in the application program. The linearization is performed, and the temperature acquired by the thermocouple/mux card is available for control strategy use or display in degrees Celsius. The conversion to units other than degrees C (Fahrenheit, Kelvin, etc.) can be accomplished by use of a calculation scaling factor.

To perform thermocouple measurement:

1. Properly configure the DAS card to be used
2. Connect the thermocouple(s) to the PCLD-8115 terminals
3. Enable the CJC circuit, and always set at channel 0 on the PCLD-8115. Of course, the CJC channel cannot be used for any other purpose during temperature measurement.

4. Select the appropriate configuration in the driver configuration dialog box - base address, etc..
5. Select the input range -0.05V to +0.05V in the application software for all thermocouple type
6. When THERMOCOUPLE TYPE in the application software is selected, the driver will perform the appropriate linearization for the selected thermocouple type with respect to any selected A/D range. However, the optimum range is the A/D range that can handle the entire temperature range for each supported thermocouple type.

DRV_AlarmConfig

```
status = DRV_AlarmConfig(DriverHandle,lpAlarmConfig)
```

Purpose

Configures the high and low limit value of the specified channel for alarm monitoring. This function only supports ADAM modules.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAlarmConfig	Input/Output	long pointer to **PT_AlarmConfig**	default	the storage address for chan, Lolimit and HiLimit

Table 5-66: DRV_AlarmConfig Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChannel** if chan is out of range
4. **CommTransmitFailed** if COMWrite failed
5. **CommReadFailed** if there is no response
6. **CommReceiveFailed** if the response string is incorrect

Note

The high and low limit values can be configured either in the ADAM utility or by within this function. By using this function, the limit values can be changed at run-time. However, this operation would take a maximum of 4 seconds.

DRV_AlarmEnable

```
status = DRV_AlarmEnable(DriverHandle,lpAlarmEnable)
```

Purpose

Enables the alarm in either momentary or latching mode. This function only supports ADAM modules.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAlarmEnable	Input/Output	long pointer to **PT_AlarmEnab le**	default	the storage address for chan, LatchMode and enabled

Table 5-67: DRV_AlarmEnable Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChannel** if chan is out of range
4. **CommTransmitFailed** if COMWrite failed
5. **CommReadFailed** if there is no response
6. **CommReceiveFailed** if the response string is incorrect

DRV_AlarmCheck

```
status = DRV_AlarmCheck(DriverHandle,lpAlarmCheck)
```

Purpose

Checks the alarm status of the specified channel.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAlarmCheck	Input/Output	long pointer to **PT_AlarmCheck**	default	the storage address for chan, LoState and HiState

Table 5-68: DRV_AlarmCheck Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChannel** if chan is out of range
4. **CommTransmitFailed** if COMWrite failed
5. **CommReadFailed** if there is no response
6. **CommReceiveFailed** if the response string is incorrect

DRV_AlarmReset

```
status = DRV_AlarmReset(DriverHandle,chan)
```

Purpose

Resets the alarm monitoring of the specified channel.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
chan	Input	long	default	channel

Table 5-69: DRV_AlarmReset Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidChannel** if chan is out of range
4. **CommTransmitFailed** if COMWrite failed
5. **CommReadFailed** if there is no response
6. **CommReceiveFailed** if the response string is incorrect

COMOpen

```
status = COMOpen(PortNum,CommID,DeviceHandle)
```

Purpose

Opens 1 of 4 serial communication ports (9 serial ports if SuperCom is installed). This function must be called before using any other communication functions.

Parameters

Name	Direction	Type	Description
PortNum	Input	unsigned short	communication port
CommID	output	long pointer to integer	port ID obtained from OpenComm
DeviceHandle	Input	long	assigned by DeviceOpen

Table 5-70: COMOpen Parameter Table

Return:

1. **SUCCESS** if successful
2. **ConfigDataLost** if configuration data is lost
3. **MemoryAllocateFailed** if memory is not enough

Note

This function calls Windows API, OpenComm, to open communication port.

COMClose

```
status = COMClose(DeviceHandle)
```

Purpose

Closes the serial port that is opened by ***COMOpen***.

Parameters

Name	Direction	Type	Description
DeviceHandle	Input	long pointer	assigned by ***DeviceOpen***

Table 5-71: COMClose Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDeviceHandle** if DeviceHandle is NULL

Note

This function calls Windows APIs, **EscapeCommFunction** and **CloseComm**, to disconnect the connection and close the communication port .

COMGetConfig

```
status = COMGetConfig(DeviceNum,buffer)
```

Purpose

Retrieves the communication port settings configured by the device installation utility.

Parameters

Name	Direction	Type	Description
DeviceNum	Input	unsigned long	device number
buffer	Output	long pointer to DEVCONFIG_COM data structure	data buffer to store the communication port settings

Table 5-72: COMGetConfig Parameter Table

Return:

1. SUCCESS if successful
2. InvalidDeviceHandle if DeviceHandle = NULL

Note

The communication port settings are stored in GDEVCFG.INI. The DEVCONFIG_COM data structure refers to DRIVER.H.

COMSetConfig

```
status = COMSetConfig(DeviceHandle,buffer)
```

Purpose

Sets the communication port configuration.

Parameters

Name	Direction	Type	Description
DeviceHandle	Input	long	assigned by ***DeviceOpen***
buffer	Output	long pointer to DEVCONFIG_COM data structure	data buffer to store the communication port settings

Table 5-73: COMSetConfig Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDeviceHandle** if DeviceHandle = NULL

Note

The communication port settings are stored in GDEVCFG.INI. The DEVCONFIG_COM data structure refers to DRIVER.H.

COMRead

```
status = COMRead(DeviceHandle,buffer,BufferSize,TimeOut,FinalChar,ReadCount)
```

Purpose

Reads data from the specified serial port.

Parameters

Name	Direction	Type	Description
DeviceHandle	Input	long	assigned by ***DeviceOpen***
buffer	Output	long pointer to string	data buffer to store the received string
BufferSize	Input	unsigned short	buffer size in bytes
TimeOut	Input	unsigned short	waiting time for response
FinalChar	Input	char	final character
ReadCount	Output	unsigned short	read count in byte

Table 5-74: COMRead Parameter Table

Return:

1. SUCCESS if successful
2. CommReadFailed if failed or no data received

Note

This function calls Windows API, **ReadComm**, to read the communication port.

This function returns if

1. A carriage return <cr> has been received. The <cr> is not guaranteed to be the last character in the string.
2. The buffer is full. There may be some more data in the receive queue.
3. Timeout.

The string is always null-terminated upon returning.

COMWrite

```
status = COMWrite(DeviceHandle,buffer,DataLength)
```

Purpose

Writes data to the specified serial port.

Parameters

Name	Direction	Type	Description
DeviceHandle	Input	long	assigned by ***DeviceOpen***
buffer	Input	long pointer to string	data buffer to store the transmitted string
DataLength	Input	unsigned short	data length in byte

Table 5-75: COMWrite Parameter Table

Return:

1. SUCCESS if successful
2. CommTransmitFailed if the transmission failed

Note

This function calls **COMWrite232**, **COMWrite485** or **COMWrite85** function according to the transmission mode configured by device installation utility.

COMWrite232

```
status = COMWrite232(DeviceHandle,buffer,DataLength)
```

Purpose

Writes data to the specified serial port.

Parameters

Name	Direction	Type	Description
DeviceHandle	Input	long	assigned by ***DeviceOpen***
buffer	Input	long pointer to string	data buffer to store the transmitted string
DataLength	Input	unsigned short	data length in bytes

Table 5-76: COMWrite232 Parameter Table

Return:

1. SUCCESS if successful
2. CommTransmitFailed if the transmission failed

Note

This function calls Windows API, **WriteComm**, to write to the communication port.

COMEscape

```
status = COMEscape(DeviceHandle,escape)
```

Purpose

This routine provides ?scape” services to the callers.

Parameters

Name	Direction	Type	Description
DeviceHandle	Input	long	assigned by ***DeviceOpen***
escape	Input	unsigned short	the escape service type

Table 5-77: COMEscape Parameter Table

Return:

1. SUCCESS if successful

Note

The escape service type:

```
escape = 1 ——EscapeFlushInput
    = 2 ——EscapeFlushOutput
    = 3 ——EscapeSetBreak
    = 4 ——EscapeClearBreak
```

The communication port settings are stored in GDEVCFG.INI. The DEVCONFIG_COM data structure refers to DRIVER.H.

DRV_FAIIntStart

```
status = DRV_FAIIntStart(DriverHandle,lpFAIIntStart)
```

Purpose

Initiates an asynchronous, single-channel data acquisition operation with Interrupt and stores its input in an array.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAIIntStart	Input/Output	long pointer to **PT_FAIIntStart**	default	the storage address for TrigSrc, SampleRate, chan, gain, buffer, count, cyclic, IntrCount.

Table 5-78: DRV_FAIIntStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.
7. **InvalidCountNumber** have these conditions, count is zero, IntrCount > count or count is not even.
8. **IllegalSpeed** if SampleRate is out of range.
9. **InvalidEventCount** if EventCount is not IntrCount multiple.
10. **ChanConflict** if interrupt channel is conflict.
11. **OpenEventFailed** if event name opens failure.
12. **InterruptProcessFailed** if interrupt proces is failure.
13. **KeInsufficientResources** if resource is conflict with another driver.
14. **KeConInterruptFailure** if connects interrupt failure

DRV_FAIDmaStart

```
status = DRV_FAIDmaStart(DriverHandle,lpFAIDmaStart)
```

Purpose

Initiates an asynchronous, single-channel data acquisition operation with DMA and stores its input in an array.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAIDmaStart	Input/Output	long pointer to **PT_FAIDmaStart**	default	the storage address for TrigSrc, SampleRate, chan, gain, buffer, count .

Table 5-79: DRV_FAIDmaStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.
7. **InvalidCountNumber** have these conditions, count is zero ,count is not even, counter is less than 2048. (When you do analog input with DMA transfer , conversion number must be bigger than 2048. (The size of Data must be bigger than PAGE_SIZE(4K).)
8. **IllegalSpeed** if SampleRate is out of range.
9. **InvalidEventCount** if EventCount is zero.
10. **ChanConflict** if interrupt channel is conflict.
11. **OpenEventFailed** if event name opens failure.
12. **KeInsufficientResources** if resource is conflict with another driver.
13. **KeConInterruptFailure** if connects interrupt failure

DRV_FAIIntScanStart

```
status = DRV_FAIIntScanStart(DriverHandle, lpFAIIntScanStart)
```

Purpose

Initiates an asynchronous, mutiple-channel data acquisition operation with Interrupt and stores its input in an array and the gain codes for the scan channels

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAIIntScanStart	Input/Output	long pointer to **PT_FAIIntScanStart**	default	the storage address for TrigSrc, SampleRate, NumChans, StartChan, GainList, buffer, count, cyclic, IntrCount.

Table 5-80: DRV_FAIIntScanStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL, or GainList = NULL.
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.
7. **InvalidCountNumber** have these conditions, count is zero, IntrCount > count or count is not even.
8. **IllegalSpeed** if SampleRate is out of range.
9. **InvalidEventCount** if EventCount is not IntrCount multiple.
10. **ChanConflict** if interrupt channel is conflict.
11. **OpenEventFailed** if event name opens failure.
12. **InterruptProcessFailed** if interrupt proces is failure.
13. **KeInsufficientResources** if resource is conflict with another driver.
14. **KeConInterruptFailure** if connects interrupt failure

DRV_FAIDmaScanStart

```
status = DRV_FAIDmaScanStart(DriverHandle,lpFAIDmaScanStart)
```

Purpose

Initiates an asynchronous, mutiple-channel data acquisition operation with DMA and stores its input in an array and the gain codes for the scan channels

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAIDmaScanStart	Input/Output	long pointer to **PT_FAIDmaScaStart**	default	the storage address for TrigSrc, SampleRate, NumChans, StartChan, GainList, buffer, count.

Table 5-81: DRV_FAIDmaScanStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL or GainList = NULL.
4. **BoardIDNotSupported** if function doesn’t have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.
7. **InvalidCountNumber** have these conditions, count is zero, count is not even, counter is less than 2048. (When you do analog input with DMA transfer , conversion number must be bigger than 2048. (The size of data must be bigger than PAGE_SIZE(4K).)
8. **IllegalSpeed** if SampleRate is out of range.
9. **InvalidEventCount** if EventCount is zero.
10. **ChanConflict** if interrupt channel is conflict.
11. **OpenEventFailed** if event name opens failure.
12. **KeInsufficientResources** if resource is conflict with another driver.
13. **KeConInterruptFailure** if connects interrupt failure

DRV_FAITransfer

```
status = DRV_FAITransfer(DriverHandle,lpFAITransfer)
```

Purpose

Transfers the data from the buffer being used for the data acquisition operation to the specified data buffer.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAITransfer	Input/Output	long pointer to **PT_FAITransfer**	default	the storage address for ActiveBuf, DataBuffer, DataType, start, count, overrun.

Table 5-82: DRV_FAITransfer Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidCountNumber** has the kinds of condition, start point is bigger conversion number, count is bigger conversion number or count is zero.

DRV_FAICheck

```
status = DRV_FAICheck(DriverHandle,lpFAICheck)
```

Purpose

Checks if the current data acquisition is complete and return current status.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAICheck	Input/Output	long pointer to **PT_FAICheck**	default	the storage address for ActiveBuf, stopped, retrieved, overrun.

Table 5-83: DRV_FAICheck Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_FAIStop

```
status = DRV_FAIStop(DriverHandle)
```

Purpose

Cancels the current data acquisition operation and resets the hardware and software.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***

Table 5-84: DRV_FAIStop Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_FAIDualDmaStart

```
status = DRV_FAIDualDmaStart(DriverHandle,lpFAIDualDmaStart)
```

Purpose

Initiates an asynchronous, single-channel data acquisition operation with DMA and stores its input in an array.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAIDualDmaStart	Input/Output	long pointer to **PT_FAIDualDmaStart**	default	the storage address for TrigSrc, SampleRate, chan, gain, BufferA, BufferB, count.

Table 5-85: DRV_FAIDualDmaStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if BufferA = NULL or BuferB = NULL.
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.
7. **InvalidDmaChannel** if DMA channel isn't set to dual DMA mode in device installation utility.
8. **InvalidCountNumber** have these conditions, count is zero, count is not even, counter is less than 2048. (When you do analog input with DMA transfer , conversion number must be bigger than 2048. (The size of data must be bigger than PAGE_SIZE(4K).)
9. **IllegalSpeed** if SampleRate is out of range.
10. **InvalidEventCount** if EventCount is zero.
11. **ChanConflict** if interrupt channel is conflict.
12. **OpenEventFailed** if event name opens failure.
13. **KeInsufficientResources** if resource is conflict with another driver.
14. **KeConInterruptFailure** if connects interrupt failure

DRV_FAIDualDmaScanStart

```
status = DRV_FAIDualDmaScanStart(DriverHandle,lpFAIDualDmaScanStart)
```

Purpose

Initiates an asynchronous, mutiple-channel data acquisition operation with DMA and stores its input in an array and the gain codes for the scan channels

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAIDualDma ScanStart	Input/Output	long pointer to **PT_FAIDualDma ScanStart**	default	the storage address for TrigSrc, SampleRate, NumChan, StartChan, GainList, BufferA, BufferB, count.

Table 5-86: DRV_FAIDualDmaScanStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if BufferA = NULL, BufferB = NULL or GainList = NULL.
4. **BoardIDNotSupported** if function doesn’t have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.
7. **InvalidDmaChannel** if DMA channel isn’t set to dual DMA mode in device installation utility.
8. **InvalidCountNumber** have these conditions, count is zero, count is not even, counter is less than 2048. (When you do analog input with DMA transfer , conversion number must be bigger than 2048. (The size of data must be bigger than PAGE_SIZE(4K).)
9. **IllegalSpeed** if SampleRate is out of range.

10. **InvalidEventCount** if EventCount is zero.
11. **ChanConflict** if interrupt channel is conflict.
12. **OpenEventFailed** if event name opens failure.
13. **KeInsufficientResources** if resource is conflict with another driver.
14. **KeConInterruptFailure** if connects interrupt failure

DRV_FAIWatchdogConfig

```
status = DRV_FAIWatchdogConfig(DriverHandle,lpFAIWatchdogConfig)
```

Purpose

Configures the hardware to acquire data before, before and after or after the signal triggers a analog watchdog. It also configures the condition and level of the analog watchdog for each channel. This function only supports PCL-1800.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAIWatchdog Config	Input/Output	long pointer to **PT_FAIWatchdo gConfig**	default	the storage address for TrigMode, NumChans, StartChan, GainList, CondList, LevelList.

Table 5-87: DRV_FAIWatchdogConfig Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if GainList = NULL, CondList = NULL, or LevelList = NULL.
4. **BoardIDNotSupported** if function doesn’t have support in driver.
5. **InvalidGain** if gain code is incorrect.
6. **InvalidChannel** if chan is out of range.

DRV_FAIIntWatchdogStart

```
status = DRV_FAIIntWatchdogStart(DriverHandle,lpFAIIntWatchdogStart)
```

Purpose

Initiates an asynchronous data acquisition operation with analog watchdog by interrupt transfer and stores its input in an array.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAIIntWatchdogStart	Input/Output	long pointer to **PT_FAIIntWatchdogStart**	default	the storage address for TrigSrc, SampleRate, buffer, BufferSize, count, cyclic, IntrCount.

Table 5-88: DRV_FAIIntWatchdogStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL.
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidCountNumber** have these conditions, count is zero, IntrCount > count or count is not even.
6. **IllegalSpeed** if SampleRate is out of range.
7. **InvalidEventCount** if EventCount is not IntrCount multiple.
8. **ChanConflict** if interrupt channel is conflict.
9. **OpenEventFailed** if event name opens failure.
10. **InterruptProcessFailed** if interrupt proces is failure.
11. **KeInsufficientResources** if resource is conflict with another driver.
12. **KeConInterruptFailure** if connects interrupt failure

DRV_FAIDmaWatchdogStart

```
status = FAIDmaWatchdogStart(DeviceHandle,
TrigSrc,SampleRate,BufferA,BufferB,BufferSize,cyclic)
```

Purpose

Initiates an asynchronous data acquisition operation with analog watchdog by DMA transfer and stores its input in an array.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAIDmaWatchdogStart	Input/Output	long pointer to **PT_FAIDmaWatchdogStart**	default	the storage address for TrigSrc, SampleRate, BufferA, BufferB, BufferSize, buffer, count, cyclic.

Table 5-89: DRV_FAIDmaWatchdogStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if BufferA = NULL or BufferB = NULL
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidDmaChannel** if DMA channel isn't set to dual DMA mode in device installation utility.
6. **InvalidCountNumber** have these conditions, count is zero, count is not even, counter is less than 2048. (When you do analog input with DMA transfer , conversion number must be bigger than 2048. (The size of data must be bigger than PAGE_SIZE(4K).)
7. **IllegalSpeed** if SampleRate is out of range.
8. **InvalidEventCount** if EventCount is zero.
9. **ChanConflict** if interrupt channel is conflict.
10. **OpenEventFailed** if event name opens failure.
11. **KeInsufficientResources** if resource is conflict with another driver.
12. **KeConInterruptFailure** if connects interrupt failure

DRV_FAICheckWatchdog

```
status = DRV_FAICheckWatchdog(DriverHandle,lpFAICheckWatchdog)
```

Purpose

Checks if the current data acquisition with watchdog is triggered.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAIWatchdog Check	Input/Output	long pointer to **PT_FAIWatchdo gCheck**	default	the storage address for DataType,ActiveBuf, triggered, TrigChan, TrigIndex, TrigData.

Table 5-90: DRV_FAICheckWatchdog Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_AllocateDMABuffer

```
status = DRV_AllocateDMABuffer(DriverHandle,lpAllocateDMABuffer)
```

Purpose

Allocates buffer for DMA data acquisition.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpAllocateDMA Buffer	Input/Output	long pointer to **PT_AllocateDM ABuffe**r	default	the storage address for CyclicMode, RequestBufSize, ActualBufSize, Buffer.

Table 5-91: DRV_AllocateDMABuffer Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **KeInsufficientResources** if kernel-mode buffer resources are not enough.
4. **KeBuferSizeTooSmall** if request buffer size is less than PAGE_SIZE(4K).
5. **KeAllocCommBufFailure** if kernel-mode buffer allocates failure. Please you reduce the buffer size request or re-boot the system.

DRV_FreeDMABuffer

```
status = DRV_FreeDMABuffer(DriverHandle,buffer)
```

Purpose

Releases the buffer allocated by ***AllocateDMABuffer***.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
buffer	Input/Output	long pointer	default	buffer address

Table 5-92: DRV_FreeDMABuffer Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_FAOIntStart

```
status = DRV_FAOIntStart(DriverHandle,lpFAOIntStart)
```

Purpose

Initiates an asynchronous analog output operation with interrupt transfer.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAOIntStart	Input/Output	long pointer to **PT_FAOIntStart**	default	the storage address for StartChan, StopChan, buffer, count, cyclic.

Table 5-93: DRV_FAOIntStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidChannel** if chan is out of range.
6. **InvalidCountNumber** have these conditions, count is zero or count is not even.
7. **IllegalSpeed** if SampleRate is out of range.
8. **InvalidEventCount** if EventCount is zero.
9. **ChanConflict** if interrupt channel is conflict.
10. **OpenEventFailed** if event name opens failure.
11. **InterruptProcessFailed** if interrupt proces is failure.
12. **KeInsufficientResources** if resource is conflict with another driver.
13. **KeConInterruptFailure** if connects interrupt failure

DRV_FAODmaStart

```
status = DRV_FAODmaStart(DriverHandle,lpFAODmaStart)
```

Purpose

Initiates an asynchronous analog output operation with DMA transfer.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAODmaStart	Input/Output	long pointer to **PT_FAODmaStart**	default	the storage address for StartChan, StopChan, buffer, count, cyclic.

Table 5-94: DRV_FAODmaStart Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidWindowsHandle** if buffer = NULL
4. **BoardIDNotSupported** if function doesn't have support in driver.
5. **InvalidChannel** if chan is out of range.
6. **InvalidCountNumber** have these conditions, count is zero ,count is not even, counter is less than 2048. (When you do analog input with DMA transfer , conversion number must be bigger than 2048. (The size of Data must be bigger than PAGE_SIZE(4K).)
7. **IllegalSpeed** if SampleRate is out of range.
8. **InvalidEventCount** if EventCount is zero.
9. **ChanConflict** if interrupt channel is conflict.
10. **OpenEventFailed** if event name opens failure.
11. **KeInsufficientResources** if resource is conflict with another driver.
12. **KeConInterruptFailure** if connects interrupt failure

DRV_FAOScale

```
status = DRV_FAOScale(DriverHandle,lpFAOScale)
```

Purpose

Translates an array of floating-point values that represent voltages into an array of binary values that produce those voltages when the driver writes the binary array to one of the boards. This function uses the current analog output configuration settings to perform the conversions.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAOScale	Input/Output	long pointer to **PT_FAOScale**	default	the storage address for StartChan, StopChan, count, VoltArray, BinArray.

Table 5-95: DRV_FAOScale Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidCountNumber** if count is zero

DRV_FAOLoad

```
status = DRV_FAOLoad(DriverHandle,lpFAOLoad)
```

Purpose

Transfers the data from the buffer being used for the data acquisition operation to the specified data buffer.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpFAOLoad	Input/Output	long pointer to **PT_FAOLoad**	default	the storage address for ActiveBuf, DataBuffer, start, count.

Table 5-96: DRV_FAOLoad Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidCountNumber** if count is zero or start point is bigger than conversion number.

DRV_FAOCheck

```
status = DRV_FAOCheck(DriverHandle,lpFAOCheck)
```

Purpose

Checks if the current analog output is complete and return current status.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
LpFAOCheck	Input/Output	long pointer to **PT_FAOCheck**	default	the storage address for ActiveBuf, stopped, CurrentCount, HalfReady.

Table 5-97: DRV_FAOCheck Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_FAOStop

```
status = DRV_FAOStop(DriverHandle)
```

Purpose

Cancels the current analog output operation and resets the hardware and software.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***

Table 5-98: DRV_FAOStop Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL

DRV_EnableEvent

```
status = DRV_EnableEvent(DriverHandle,lpEnableEvent)
```

Purpose

Enables or disables event. This funtion supports with interrupt and DMA features.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpEnableEvent	Input/Output	long pointer to **PT_EnableEvent**	default	the storage address for EventType, Enabled, Count.

Table 5-99: DRV_EnableEvent Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **InvalidInputParam** if Count is zero.
4. **InvalidEventType** if event type is out of range.

DRV_CheckEvent

```
status = CheckEvent(DriverHandle,lpCheckEvent)
```

Purpose

Clear event and read current status.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpCheckEvent	Input/Output	long pointer to **PT_CheckEvent**	default	the storage address for EventType, Milliseconds.

Table 5-100: DRV_CheckEvent Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **EventTimeOut** if the Time-out interval elapsed in milliseconds parameter.

DRV_TimerCountSetting

```
status = DRV_TimerCountSetting(DriverHandle, lpTimerCountSetting)
```

Purpose

For PCI data acquisition and control device, the Timer informations are defined from device installation. But we provide this API to change the Counter/Timer value dynamically.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***
lpTimerCountSetting	Input/Output	long pointer to **PT_TimerCountSetting**	default	the storage address for counter and Count

Table 5-101: DRV_TimerCountSetting Parameter Table

Return

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL
3. **BoardIDNotSupported** if the function is not supported for this device
4. **InvalidChannel** if the port number is out of range

Note

If the cascade mode of PCI device is configured to be “Yes” in Device Installation Utility, then the high word of the Count is set to counter1 and the low word of the Count is set to counter0.

DRV_ClearOverrun

```
status = DRV_ClearOverrun(DriverHandle)
```

Purpose

Clear overrun flag.

Parameters

Name	Direction	Type	Range	Description
DriverHandle	Input	long	default	assigned by ***DRV_DeviceOpen***

Table 5-102: DRV_ClearOverrun Parameter Table

Return:

1. **SUCCESS** if successful
2. **InvalidDriverHandle** if DriverHandle = NULL function .

5.3 Data Structures

GAINLIST

```
typedef  struct tagGAINLIST
{
  USHORT    usGainCde;
  FLOAT     fMaxGainVal;
  FLOAT     fMinGainVal;
  CHAR      szGainStr[16];
} GAINLIST;
```

GAINLIST is used by **DRV_DeviceGetFeatures** function .

Member Description

Name	Direction	Type	Range	Description
usGainCde	output	unsigned short	default	the gain code for each voltage output range which you may also refer to User Manual
fMaxGainVal	output	floating-point	default	the maximum gain code
fMinGainVal	output	floating-point	default	the minimum gain code
szGainStr	output	array of char		the voltage output range for the range code

Table 5-103: GAINLIST Member Description

DEVFEATURES

```
typedef struct tagDEVFEATURES
{
  CHAR     szDriverVer[8];
  CHAR     szDriverName[MAX_DRIVER_NAME_LEN];
  DWORD    dwBoardID;
  USHORT   usMaxAIDiffChl;
  USHORT   usMaxAISiglChl;
  USHORT   usMaxAOChl;
  USHORT   usMaxDOChl;
  USHORT   usMaxDIChl;
  USHORT   usDIOPort;
  USHORT   usMaxTimerChl;
  USHORT   usMaxAlarmChl;
  USHORT   usNumADBit;
  USHORT   usNumADByte;
  USHORT   usNumDABit;
  USHORT   usNumDAByte;
  USHORT   usNumGain ;
  GAINLIST glGainList[16];
  DWORD    dwPermutation[4];
} DEVFEATURES, FAR * LPDEVFEATURES;
```

DEVFEATURES is used by **DRV_DeviceGetFeatures** function .

Member Description

Name	Direction	Type	Range	Description
szDriverVer	output	array of char		Device driver version
szDriverName	output	array of char		device driver name
dwBoardID	output	double word		board ID
usMaxAIDiffChl	output	unsigned short	depends on hardware	Max. number of differential channel
usMaxAISiglChl	output	unsigned short	depends on hardware	Max. number of single-end channel
usMaxAOChl	output	unsigned short	depends on hardware	Max. number of analog out channel
usMaxDOChl	output	unsigned short	depends on hardware	Max. number of digital out channel
usMaxDIChl	output	unsigned short	depends on hardware	Max. number of digital input channel
usMaxTimerChl	output	unsigned short	depends on hardware	Max. number of Counter/Timer channel
usMaxAlarmChl	output	unsigned short	depends on hardware	Max number of alarm channel
usNumADBit	output	unsigned short	depends on hardware	number of bits for A/D converter
usNumADByte	output	unsigned short	depends on hardware	A/D channel width in bytes
usNumDABit	output	unsigned short	depends on hardware	number of bits for D/A converter
usNumDAByte	output	unsigned short	depends on hardware	D/A channel width in bytes
usNumGain	output	unsigned short	depends on hardware	Max. number of gain code
glGainList	output	array of GAINLIST	depends on hardware	Gain listing
dwPermutation	output	double word	depends on hardware	Permutation

Table 5-104: DEVFEATURES Member Description

Note

Definition for **dwPermutation** member

```
Bit 0: Software AI
Bit 1: DMA AI
Bit 2: Interrupt AI
Bit 3: Condition AI
Bit 4: Software AO
Bit 5: DMA AO
Bit 6: Interrupt AO
Bit 7: Condition AO
Bit 8: Software DI
Bit 9: DMA DI
Bit 10: Interrupt DI
Bit 11: Condition DI
Bit 12: Software DO
Bit 13: DMA DO
Bit 14: Interrupt DO
Bit 15: Condition DO
Bit 16: High Gain
Bit 17: Auto Channel Scan
Bit 18: Pacer Trigger
Bit 19: External Trigger
Bit 20: Down Counter
Bit 21: Dual DMA
Bit 22: Monitoring
Bit 23: QCounter
```

AOSET

```
typedef  struct tagAOSET
{
  USHORT  usAOSource;
  FLOAT   fAOMaxVol;
  FLOAT   fAOMinVol;
} AOSET, FAR * LPAOSET;
```

AOSET Member Description

Name	Direction	Type	Range	Description
usAOSource	input	unsigned short	**0-** internal **1-** external	
fAOMaxVol	input	floating-point	depend on hardware	Max. output voltage
fAOMinVol	input	floating-point	depend on hardware	Min. output voltage

Table 5-105: AOSET Member Description

DAUGHTERSET

```
typedef  struct tagDAUGHTERSET
{
  DWORD     dwBoardID;
  USHORT    usNum;
  FLOAT     fGain;
  USHORT    usCards;
} DAUGHTERSET, FAR * LPDAUGHTERSET;
```

DAUGHTERSET is used by **DRV_AIGetConfig** function.

Member Description

Name	Direction	Type	Range	Description
dwBoardID	input	double word	Refer to Driver.H	expansion board ID
usNum	input	unsigned short	depend on device installation	available expansion channels
fGain	input	floating-point	depend on hardware	gain for expansion channel
usCards	input	unsigned short	depend on device installation	number of expansion cards

Table 5-106: DAUGHTERSET Member Description

DEVCONFIG_AI

```
typedef struct tagDEVCONFIG_AI
{
  DWORD     dwBoardID;
  USHORT    usChanConfig;
  USHORT    usGainCtrMode;
  USHORT    usPolarity;
  USHORT    usDasGain;
  USHORT    usNumExpChan;
  USHORT    usCjcChannel;
  DAUGHTERSET Daughter[MAX_DAUGHTER_NUM];
} DEVCONFIG_AI, FAR * LPDEVCONFIG_AI;
```

DEVCONFIG_AI is used by **DRV_AIGetConfig** function.

Member Description

Name	Direction	Type	Range	Description
dwBoardID	input	double word	Refer to **Driver.H**	board ID code
usChanConfig	input	unsigned short	**0**-single ended **1**-differential	Analog input mode
usGainCtrMode	input	unsigned short	**1**-by jumper **0-**programmable	
usPolarity	input	unsigned short	**0**-bipolar **1**-unipolar	
usDasGain	input	unsigned short	refer to gain list in user s manual	the gain code of the channel on DAS card
usNumExpChan	input	unsigned short	depend on device installation	DAS channels attached expansion board
usCjcChannel	input	unsigned short		cold junction channel

Table 5-107: DEVCONFIG_AI Member Description

DEVCONFIG_COM

```
typedef struct tagDEVCONFIG_COM
{
  USHORT  usCommPort;
  DWORD   dwBaudRate;
  USHORT  usParity;
  USHORT  usDataBits;
  USHORT  usStopBits;
  USHORT  usTxMode;
  USHORT  usPortAddress;
} DEVCONFIG_COM, FAR * LPDEVCONFIG_COM;
```

DEVCONFIG_COM is used by **COMGetConfig** and **COMSetConfig** functions.

Member Description

Name	Direction	Type	Range	Description
usCommPort	input	unsigned short		serial port
dwBaudRate	input	double word	actual value	baud rate
usParity	input	unsigned short		parity check
usDataBits	input	unsigned short		data bits
usStopBits	input	unsigned short		stop bits
usTxMode	input	unsigned short		transmission mode
usPortAddress	input	unsigned short		communication port address

Table 5-108: DEVCONFIG_COM Member Description

Note:

For example, dwBaudRate = 9600 when sets the baud-rate of COM port in 9600 BPS. Please also refer to DCB (Device-Control Block) structure in Win32 SDK help for more information.

TRIGLEVEL

```
typedef struct tagTRIGLEVEL
{
  FLOAT fLow;
  FLOAT fHigh;
} TRIGLEVEL;
```

TRIGLEVEL is used by DAQConfigWatchdogfunction.

Member Description

Name	Direction	Type	Range	Description
fLow	input	floating-point		
fHigh	input	floating-point		

Table 5-109: TRIGLEVEL Member Description

PT_EVLIST, DEVLIST

```
typedef struct tagPT_DEVLIST
{
  DWORD     dwDeviceNum;
  char      szDeviceName[50];
  SHORT     nNumOfSubdevices;
}PT_DEVLIST;
```

PT_DEVLIST is used by **DRV_DeviceGetList** function.

Member Description

Name	Direction	Type	Range	Description
dwDeviceNum	Input	DWORD	0 ~ 999	Device number
szDeviceName[50]	Output	char		Device name
nNumOfSubdevices	Output	short		Number of sub devices in device number

Table 5-110: PT_EVLIST, DEVLIST Member Description

PT_DeviceGetFeatures

```
typedef struct tagPT_DeviceGetFeatures
{
  LPDEVFEATURES       buffer;
  USHORT    size;
}PT_DeviceGetFeatures;
```

PT_DeviceGetFeatures is used by **DRV_DeviceGetFeatures** function.

Member Description

Name	Direction	Type	Range	Description
buffer	Output	long pointer to DEVFEATURES		the storage address of the device features
size	Input	unsigned short		buffer size

Table 5-111: PT_DeviceGetFeatures Member Description

PT_AIConfig

```
typedef struct tagPT_AIConfig
{
  USHORT    DasChan;
  USHORT    DasGain;
} PT_AIConfig, FAR * LPT_AIConfig;
```

PT_AIConfig is used by **DRV_AIConfig** function.

Member Description

Name	Direction	Type	Range	Description
DasChan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
DasGain	Input	unsigned short	refer to gain list in hardware manual	gain code

Table 5-112: PT_AIConfig Member Description

PT_AIGetConfig

```
typedef struct tagPT_AIGetConfig
{
  LPDEVCONFIG_AI buffer;
  USHORT    size;
} PT_AIGetConfig, FAR * LPT_AIGetConfig;
```

PT_AIGetConfig is used by **DRV_AIGetConfig** function.

Member Description

Name	Direction	Type	Range	Description
buffer	Output	long pointer to DEVCONFIG_AI		the storage address of the AI configuration
size	Input	unsigned short		buffer size

Table 5-113: PT_AIGetConfig Member Description

PT_AIBinaryIn

```
typedef struct tagPT_AIBinaryIn
{
  USHORT    chan;
  USHORT    TrigMode;
  USHORT far  *reading;
 } PT_AIBinaryIn, FAR * LPT_AIBinaryIn;
```

PT_AIBinaryIn is used by **DRV_AIBinaryIn** function.

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
TrigMode	Input	unsigned short	**0**-normal (software) **1**-external	trigger mode
reading	Output	long pointer to unsigned short	depends on hardware	raw data reading from the sampled channel

Table 5-114: PT_AIBinaryIn Member Description

PT_AIScale

```
typedef struct tagPT_AIScale
{
  USHORT    reading;
  FLOAT     MaxVolt;
  USHORT    offset;
  FLOAT far   *voltage;
} PT_AIScale, FAR * LPT_AIScale;
```

PT_AIScale is used by **DRV_AIScale** function .

Member Description

Name	Direction	Type	Range	Description
reading	Input	unsigned short		result of AIBinaryIn
MaxVolt	Input	float		max. voltage
MaxCount	Input	unsigned short		max. count
offset	Input	unsigned short		binary offset for zero voltage
voltage	Output	long pointer to floating-point	-MaxVolt to +MaxVolt	computed floating-point voltage

Table 5-115: PT_AIScale Member Description

PT_AIVoltageIn

```
typedef struct tagPT_AIVoltageIn
{
  USHORT    chan;
  USHORT    gain;
  USHORT    TrigMode;
  FLOAT far   *voltage;
} PT_AIVoltageIn, FAR * LPT_AIVoltageIn;
```

PT_AIVoltageIn is used by **DRV_AIVoltageIn** function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
gain	Input	unsigned short	refer to gain list in hardware manual	gain code
TrigMode	Input	unsigned short	**0**-normal (software) **1**-external	trigger mode
voltage	Output	long pointer to floating-point	depends on input range	the measured voltage returned, scaled to units of volts

Table 5-116: PT_AIVoltageIn Member Description

PT_AIVoltageInExp

```
typedef struct tagPT_AIVoltageInExp
{
  USHORT    DasChan;
  USHORT    DasGain;
  USHORT    ExpChan;
  FLOAT far   *voltage;
} PT_AIVoltageInExp, FAR * LPT_AIVoltageInExp;
```

PT_AIVoltageInExp is used by **DRV_AIVoltageInExp** function.

Member Description

Name	Direction	Type	Range	Description
DasChan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel on the DASCard
DasGain	Input	unsigned short	refer to gain list in hardware manual	the gain code of the channel on DAS card
ExpChan	Input	unsigned short	see below	the sampled channel on the expansion board
voltage	Output	long pointer to floating-point	depends on input range	the measured voltage returned, scaled to units of volts

Table 5-117: PT_AIVoltageInExp Member Description

PT_MAIConfig

```
typedef struct tagPT_MAIConfig
{
  USHORT    NumChan;
  USHORT    StartChan;
  USHORT far  *GainArray;
} PT_MAIConfig, FAR * LPT_MAIConfig;
```

PT_MAIConfig is used by **DRV_MAIConfig** function .

Member Description

Name	Direction	Type	Range	Description
NumChan	Input	unsigned short	1-n (n depends on hardware)	number of channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	the start one of scan channels
GainArray	Input	long pointer to unsigned short array	refer to gain code list in hardware manual	gain code list

Table 5-118: PT_MAIConfig Member Description

PT_MAIBinaryIn

```
typedef struct tagPT_MAIBinaryIn
{
  USHORT    NumChan;
  USHORT    StartChan;
  USHORT    TrigMode;
  USHORT far  *ReadingArray;
} PT_MAIBinaryIn, FAR * LPT_MAIBinaryIn;
```

PT_MAIBinaryIn is used by **DRV_MAIBinaryIn** function .

Member Description

Name	Direction	Type	Range	Description
NumChan	Input	unsigned short	1-n (n depends on hardware)	number of the channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	start one of scan channels
TrigMode	Input	unsigned short	**0**-normal (software) **1**-external	trigger mode
ReadingArray	Output	long pointer to unsigned short array	depends on hardware	the unscaled result

Table 5-119: PT_MAIBinaryIn Member Description

PT_MAIVoltageIn

```
typedef struct tagPT_MAIVoltageIn
{
  USHORT    NumChan;
  USHORT    StartChan;
  USHORT far  *GainArray;
  USHORT    TrigMode;
  FLOAT far   *VoltageArray;
} PT_MAIVoltageIn, FAR * LPT_MAIVoltageIn;
```

PT_MAIVoltageIn is used by **DRV_MAIVoltageIn** function.

Member Description

Name	Direction	Type	Range	Description
NumChan	Input	unsigned short	1-n (n depends on hardware)	number of the channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	the start one of scan channels
GainArray	Input	long pointer to unsigned short	refer to gain list in hardware manual	gain code array
TrigMode	Input	unsigned short	0-normal (software) 1-external	trigger mode
VoltageArray	Output	long pointer to floating-point array	depends on input range	the measured voltages returned, scaled to units of volts

Table 5-120: PT_MAIVoltageIn Member Description

PT_MAIVoltageInExp

```
typedef struct tagPT_MAIVoltageInExp
{
  USHORT              NumChan;
  USHORT far          *DasChanArray;
  USHORT far          *DasGainArray;
  USHORT far          *ExpChanArray;
  FLOAT far *VoltageArray;
} PT_MAIVoltageInExp, FAR * LPT_MAIVoltageInExp;
```

PT_MAIVoltageInExp is used by **DRV_MAIVoltageInExp** function .

Member Description

Name	Direction	Type	Range	Description
NumChan	Input	unsigned short	1-n (n depends on hardware)	the number of channels
DasChanArray	Input	long pointer to unsigned short array	0-n (n depends on hardware)	scan channels on DAS card
DasGainArray	Input	long pointer to unsigned short array	refer to gain list in hardware manual	gain array for DAS card
ExpChanArray	Input	long pointer to unsigned short array	see below	the sampled channels on the expansion board
VoltageArray	Output	long pointer to floating-point array	depends on input range	the measured voltages returned, scaled to units of volts

Table 5-121: PT_MAIVoltageInExp Member Description

PT_TCMuxRead

```
typedef struct tagPT_TCMuxRead
{
  USHORT    DasChan;
  USHORT    DasGain;
  USHORT    ExpChan;
  USHORT    TCType;
  USHORT    TempScale;
  FLOAT far   *temp;
} PT_TCMuxRead, FAR * LPT_TCMuxRead;
```

PT_TCMuxRead is used by **DRV_TCMuxRead** function.

Member Description

Name	Direction	Type	Range	Description
DasChan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
DasGain	Input	unsigned short	see below	gain code of the sampled channel on DASCard
ExpChan	Input	unsigned short	see below	the thermocouple channel on the expansion board
TCType	Input	unsigned short	0,1,2,3,4,5,6	thermocouple type: J (0), K (1), S (2), T (3), B (4), R (5), E (6)
TempScale	Input	unsigned short	0,1,2,3	temperature unit: Celsius (0), Fahrenheit (1), Rankine(2), Kelvin (3)
temp	Output	long pointer to floating-point	depends on hardware	measured temperature

Table 5-122: PT_TCMuxRead Member Description

PT_AOConfig

```
typedef struct tagPT_AOConfig
{
  USHORT    chan;
  USHORT    RefSrc;
  FLOAT     MaxValue;
  FLOAT     MinValue;
} PT_AOConfig, FAR * LPT_AOConfig;
```

PT_AOConfig DRV_AOConfig function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the output channel
RefSrc	Input	unsigned short	0 or 1	reference source: internal (0) or external (1)
MaxValue	Input	floating-point	depends on hardware	max. reference voltage
MinValue	Input	floating-point	depends on hardware	min. reference voltage

Table 5-123: PT_AOConfig Member Description

PT_AOBinaryOut

```
typedef struct tagPT_AOBinaryOut
{
  USHORT    chan;
  USHORT    BinData;
} PT_AOBinaryOut, FAR * LPT_AOBinaryOut;
```

PT_AOBinaryOut is used by **DRV_AOBinaryOut** function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the output channel
BinData	Input	unsigned short	depends on hardware	digital value to be written

Table 5-124: PT_AOBinaryOut Member Description

PT_AOVoltageOut

```
typedef struct tagPT_AOVoltageOut
{
  USHORT    chan;
  FLOAT     OutputValue;
} PT_AOVoltageOut, FAR * LPT_AOVoltageOut;
```

PT_AOVoltageOut is used by **DRV_AOVoltageOut** function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the output channel
OutputValue	Input	floating-point	Min. DA Ref. ≤ **voltage** ≤ Max. DA Ref.	floating-point value to be scaled and written

Table 5-125: PT_AOVoltageOut Member Description

PT_AOScale

```
typedef struct tagPT_AOScale
{
  USHORT    chan;
  FLOAT     OutputValue;
  USHORT far  *BinData;
}PT_AOScale,    FAR * LPT_AOScale;
```

PT_AOScale is used by **DRV_AOScale** function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the output channel
OutputValue	Input	floating	Min. DA Ref. ≤ **voltage** ≤ Max. DA Ref.	floating value to be scaled
BinData	Output	long pointer to unsigned short	depends on hardware	converted binary value returned

Table 5-126: PT_AOScale Member Description

PT_AOCurrentOut

```
typedef struct tagPT_AOCurrentOut
{
    USHORT      chan;          // AO Out channel
FLOAT           OutputValue;   // Output Current value
} PT_AOCurrentOut, FAR * LPT_AOCurrentOut;
```

PT_AOCurrentOut is used by **DRV_AOCurrentOut** function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-n (n depends on hardware)	the output channel
OutputValue	Input	floating	Min. DA Ref. ≤ **voltage** ≤ Max. DA Ref.	floating value to be scaled

Table 5-127: PT_AOCurrentOut Member Description

PT_DioSetPortMode

```
typedef struct tagPT_DioSetPortMode
{
  SHORT     port;
  USHORT    dir;
} PT_DioSetPortMode, FAR * LPT_DioSetPortMode;
```

PT_DioSetPortMode is used by **DRV_DioSetPortMode** function .

Member Description

Name	Direction	Type	Range	Description
port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
dir	Input	unsigned short	0 or 1	direction: input (0) or output(1)

Table 5-128: PT_DioSetPortMode Member Description

PT_DioGetConfig

```
typedef struct tagPT_DioGetConfig
{
  SHORT far *PortArray;
  USHORT    NumOfPorts;
} PT_DioGetConfig, FAR * LPT_DioGetConfig;
```

PT_DioGetConfig is used by **DRV_DioGetConfig** function .

Member Description

Name	Direction	Type	Range	Description
PortArray	Output	short pointer	Default	the storage address of port direction.
NumOfPorts	Input	unsigned short	Default	number of ports

Table 5-129: PT_DioGetConfig Member Description

Note

This structure just supports the board which emulates the digital input/output operation in 8255 Mode 0.   (for examples : PCL-722/724/731, PCL-814-DIO-1). The **PortArray** can be set to below values:

1. Input mode : **0**
2. Output mode : **1**
3. Input / Output mode : **2**
4. **Low nibble**(PC0~PC3) as input mode and **High nibble** (PC4~PC7) as output mode : **3**
5. **Low nibble**(PC0~PC3) as output mode and **High nibble** (PC4~PC7) as input mode : **4**

PT_DioReadPortByte

```
typedef struct tagPT_DioReadPortByte
{
  USHORT      port;
  USHORT far  *value;
} PT_DioReadPortByte, FAR * LPT_DioReadPortByte;
```

PT_DioReadPortByte is used by **DRV_DioReadPortByte** function .

Member Description

Name	Direction	Type	Range	Description
port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
value	Output	long pointer to unsigned short	default	8-bit digital data read from the specified port

Table 5-130: PT_DioReadPortByte Member Description

PT_DioWritePortByte

```
typedef struct tagPT_DioWritePortByte
{
  USHORT    port;
  USHORT    mask;
  USHORT    state;
} PT_DioWritePortByte, FAR * LPT_DioWritePortByte;
```

PT_DioWritePortByte is used by **DRV_DioWritePortByte** function
.

Member Description

Name	Direction	Type	Range	Description
port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
mask	Input	unsigned short	default	specifies which bit(s) of data should be sent to the digital output port and which bits remain unchanged
state	Input	unsigned short	default	new digital logic state

Table 5-131: PT_DioWritePortByte Member Description

PT_DioReadBit

```
typedef struct tagPT_DioReadBit
{
  USHORT    port;
  USHORT    bit;
  USHORT far  *state;
} PT_DioReadBit, FAR * LPT_DioReadBit;
```

PT_DioReadBit is used by **DRV_DioReadBit** function .

Member Description

Name	Direction	Type	Range	Description
port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
bit	Input	unsigned short	0-7	the bit number
state	Output	long pointer to unsigned short	0 or 1	bit data read from the specified port

Table 5-132: PT_DioReadBit Member Description

PT_DioWriteBit

```
typedef struct tagPT_DioWriteBit
{
  USHORT      port;
  USHORT      bit;
  USHORT      state;
} PT_DioWriteBit, FAR * LPT_DioWriteBit;
```

PT_DioWriteBit is used by **DRV_DioWriteBit** function .

Member Description

Name	Direction	Type	Range	Description
Port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
Bit	Input	unsigned short	0-7	the bit number
State	Input	unsigned short	default	new digital logic state

Table 5-133: PT_DioWriteBit Member Description

PT_DioGetCurrentDOByte

```
typedef struct tagPT_DioGetCurrentDOByte
{
  USHORT    port;
  USHORT far  *value;
} PT_DioGetCurrentDOByte, FAR * LPT_DioGetCurrentDOByte;
```

PT_DioGetCurrentDOByte is used by **DRV_DioGetCurrentDOByte** function .

Member Description

Name	Direction	Type	Range	Description
port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
value	Output	long pointer to unsigned short	default	8-bit digital data read from the specified port

Table 5-134: PT_DioGetCurrentDOByte Member Description

PT_DioGetCurrentDOBit

```
typedef struct tagPT_DioGetCurrentDOBit
{
  USHORT    port;
  USHORT    bit;
  USHORT far  *state;
} PT_DioGetCurrentDOBit, FAR * LPT_DioGetCurrentDOBit;
```

PT_DioGetCurrentDOBit is used by **DRV_DioGetCurrentDOBit** function .

Member Description

Name	Direction	Type	Range	Description
port	Input	unsigned short	0-n (n depends on hardware)	the digital port number
bit	Input	unsigned short	0-7	the bit number
state	Output	long pointer to unsigned short	0 or 1	bit data read from the specified port

Table 5-135: PT_DioGetCurrentDOBit Member Description

PT_WritePortByte

```
typedef struct tagPT_WritePortByte
{
  USHORT    port;
  USHORT    ByteData;
} PT_WritePortByte, FAR * LPT_WritePortByte;
```

PT_WritePortByte is used by **DRV_WritePortByte** function .

Member Description

Name	Direction	Type	Description
port	Input	unsigned short	I/O port address
ByteData	Input	unsigned short	8-bit output data

Table 5-136: PT_WritePortByte Member Description

Remarks

- Digital I/O channel 0 – 7 ➔ Port 0
- Digital I/O channel 8 – 15 ➔ Port 1
- Digital I/O channel 9 – 23 ➔ Port 2
- Digital I/O channel 24 – 31 ➔ Port 3

PT_WritePortWord

```
typedef struct tagPT_WritePortWord
{
  USHORT    port;
  USHORT    WordData;
} PT_WritePortWord, FAR * LPT_WritePortWord;
```

PT_WritePortWord is used by **DRV_WritePortWord** function .

Member Description

Name	Direction	Type	Description
port	Input	unsigned short	I/O port address
WordData	Input	unsigned short	16-bit output data

Table 5-137: PT_WritePortWord Member Description

PT_ReadPortByte

```
typedef struct tagPT_ReadPortByte
{
  USHORT    port;
  USHORT far  *ByteData;
} PT_ReadPortByte, FAR * LPT_ReadPortByte;
```

PT_ReadPortByte is used by **DRV_ReadPortByte** function .

Member Description

Name	Direction	Type	Description
port	Input	unsigned short	I/O port address
ByteData	Output	long pointer to unsigned short	8-bit input data

Table 5-138: PT_ReadPortByte Member Description

PT_ReadPortWord

```
typedef struct tagPT_ReadPortWord
{
  USHORT    port;
  USHORT far  *WordData;
} PT_ReadPortWord, FAR * LPT_ReadPortWord;
```

PT_ReadPortWord is used by **DRV_ReadPortWord** function .

Member Description

Name	Direction	Type	Description
port	Input	unsigned short	I/O port address
WordData	Output	long pointer to unsigned short	16-bit input data

Table 5-139: PT_ReadPortWord Member Description

PT_CounterEventStart

```
typedef struct tagPT_CounterEventStart
{
  USHORT    counter;
  USHORT    GateMode;
} PT_CounterEventStart, FAR * LPT_CounterEventStart;
```

PT_CounterEventStart is used by **DRV_CounterEventStart** function .

Member Description

Name	Direction	Type	Range	Description
counter	Input	unsigned short	0-n (n depends on hardware)	counter number
GateMode	Input	unsigned short	0,1,2,3,4	gating mode to be used for AMD Am9513A

Table 5-140: PT_CounterEventStart Member Description

PT_CounterEventRead

```
typedef struct tagPT_CounterEventRead
{
  USHORT    counter;
  USHORT far  *overflow;
  ULONG far   *count;
} PT_CounterEventRead, FAR * LPT_CounterEventRead;
```

PT_CounterEventRead is used by **DRV_CounterEventRead** function .

Member Description

Name	Direction	Type	Description
counter	Input	unsigned short	counter number
overflow	Output	long pointer to unsigned short	overflow state of the counter, 1 means overflow, otherwise 0
count	Output	long pointer to unsigned long	current total of the specified counter

Table 5-141: PT_CounterEventRead Member Description

PT_CounterFreqStart

```
typedef struct tagPT_CounterFreqStart
{
  USHORT    counter;
  USHORT    GatePeriod;
  USHORT    GateMode;
} PT_CounterFreqStart, FAR * LPT_CounterFreqStart;
```

PT_CounterFreqStart is used by **DRV_CounterFreqStart** function
.

Member Description

Name	Direction	Type	Description
counter	Input	unsigned short	counter number
GatePeriod	Input	unsigned short	gating period in seconds for AMD Am9513A
GateMode	Input	unsigned short	gating mode to be used for AMD Am9513A

Table 5-142: PT_CounterFreqStart Member Description

PT_CounterFreqRead

```
typedef struct tagPT_CounterFreqRead
{
  USHORT    counter;
  FLOAT far   *freq;
} PT_CounterFreqRead, FAR * LPT_CounterFreqRead;
```

PT_CounterFreqRead is used by **DRV_CounterFreqRead** function
.

Member Description

Name	Direction	Type	Description
counter	Input	unsigned short	counter number
freq	Output	long pointer to floating-point	current frequency returned

Table 5-143: PT_CounterFreqRead Member Description

PT_CounterPulseStart

```
typedef struct tagPT_CounterPulseStart
{
  USHORT    counter;
  FLOAT     period;
  FLOAT     UpCycle;
  USHORT    GateMode;
} PT_CounterPulseStart, FAR * LPT_CounterPulseStart;
```

PT_CounterPulseStart is used by **DRV_CounterPulseStart** function .

Member Description

Name	Direction	Type	Description
Counter	Input	unsigned short	counter number
Period	Input	floating-point	total period in seconds
UpCycle	Input	floating-point	the first 1/2 cycle length in seconds for AMD Am9513A
GateMode	Input	unsigned short	~~gating mode to be used for AMD Am9513A~~

Table 5-144: PT_CounterPulseStart Member Description

PT_QCounterConfig

```
typedef struct tagPT_QCounterConfig
{
  USHORT    counter;
  USHORT    LatchSrc;
  USHORT    LatchOverflow;
  USHORT    ResetOnLatch;
  USHORT    ResetValue;
} PT_QCounterConfig, FAR * LPT_QCounterConfig;
```

PT_QCounterConfig is used by **DRV_QCounterConfig** function .

Member Description

Name	Direction	Type	Description
counter	Input	unsigned short	counter number
LatchSrc	Input	unsigned short	latch source, you can set the following value : **SWLATCH**(0), **INDEXINLATCH**(1), **DI0LACTCH**(2), **DI1LACTCH**(3), **TIMERLATCH**(4).
LatchOverflow	Input	unsigned short	free run(0) or latch on overflow(1)
ResetOnLatch	Input	unsigned short	reset after counter is latched(1), otherwise(0)
ResetValue	Input	unsigned short	Set reset value to 000000 (0) , 800000(1)

Table 5-145: PT_QCounterConfig Member Description

PT_QCounterConfigSys

```
typedef struct tagPT_QCounterConfigSys
{
  USHORT    SysClock;
  USHORT    TimeBase;
  USHORT    TimeDivider;
  USHORT    CascadeMode;
} PT_QCounterConfigSys, FAR * LPT_QCounterConfigSys;
```

PT_QCounterConfigSys is used by **DRV_QCounterConfigSys** function .

Member Description

Name	Direction	Type	Description
DeviceHandle	Input	long	assigned by ***DeviceOpen***
SysClock	Input	unsigned short	clock frequency for digital filter, you can set **SYS8MHZ**(0), **SYS4MHZ**(1), **SYS2MHZ**(2)
TimeBase	Input	unsigned short	16C54 time base control. **TPOINT1MS**(0)=0.1 ms, **T1MS**(1)=1ms, **T10MS**(2)= 10ms, **T100MS**(3)=100ms, **T10000MS**(4)=1second
TimeDivider	Input	unsigned short	divider control value
CascadeMode	Input	unsigned short	cascade mode **NOCASCADE**(0), **CASCADE**(1)

Table 5-146: PT_QCounterConfigSys Member Description

PT_QCounterStart

```
typedef struct tagPT_QCounterStart
{
  USHORT    counter;
  USHORT    InputMode;
} PT_QCounterStart, FAR * LPT_QCounterStart;
```

PT_QCounterStart is used by **DRV_QCounterStart** function .

Member Description

Name	Direction	Type	Description
counter	Input	unsigned short	counter number
InputMode	Input	unsigned short	Input mode control **DISABLE**(0), **ABPHASEX1**(1) **ABPHASEX2**(2) **ABPHASEX4**(3) **TWOPULSEIN**(4) **ONEPULSEIN**(5)

Table 5-147: PT_QCounterStart Member Description

PT_QCounterRead

```
typedef struct tagPT_QCounterRead
{
  USHORT    counter;
  USHORT far  *overflow;
  ULONG far   *LoCount;
  ULONG far   *HiCount;
} PT_QCounterRead, FAR * LPT_QCounterRead;
```

PT_QCounterRead is used by **DRV_QCounterRead** function .

Member Description

Name	Direction	Type	Description
Counter	Input	unsigned short	counter number
Overflow	Output	long pointer to unsigned short	overflow state of the counter, 1 means overflow, otherwise 0
LoCount	Output	long pointer to unsigned long	the low 32-bit data of current total
HiCount	Output	long pointer to unsigned long	the high 32-bit of current total

Table 5-148: PT_QCounterRead Member Description

PT_TimerCountSetting

```
typedef struct tagPT_TimerCountSetting
{
  USHORT    counter;
  ULONG     Count;
} PT_TimerCountSetting, FAR * LPT_TimerCountSetting;
```

PT_TimerCountSetting is used by **DRV_ TimerCountSetting** function .

Member Description

Name	Direction	Type	Description
counter	Input	unsigned short	counter number of PCI device. (For PCI-1750, there are counter0, counter1 and counter2)
Count	Input	unsigned long	user input value for Timer count setting.

Table 5-149: PT_TimerCountSetting Member Description

PT_DICounter

```
  typedef struct tagPT_DICounter
  {
USHORT          usEventType;
     USHORT     usEventEnable;
     USHORT     usCount;
     USHORT     usEnable;
     USHORT     usTrigEdge;
     USHORT far * usPreset;
     USHORT     usMatchEnable;
     USHORT far * usValue;
     USHORT     usOverflow;
     USHORT     usDirection;
  } PT_DICounter, FAR * LPT_DICounter;
```

PT_DICounter is used by **DRV_EnableEventEx** function .

Member Description

Name	Direction	Type	Description
usEventType	Input	unsigned short	event type
usEventEnable	Input	unsigned short	event enable/disable bit
usCount	Input	unsigned short	reserved
usEnable	Input	unsigned short	counter0 ~ counter7 enable/disable setting. For example, if bit0 = 1, counter0 is enabled.
usTrigEdge	Input	unsigned short	counter trigger edge, 0: rising edge, 1:falling edge
usPreset	Input	long pointer to unsigned short	counter pre_setting value
usMatchEnable	Input	unsigned short	the match function of counter0 ~ counter7 enable/disable setting. For example, if bit0 = 1, the match function of counter0 is enabled.
usValue	Input	long pointer to unsigned short	counter match value
usOverflow	Input	unsigned short	counter overflow data
usDirection	Input	unsigned short	up/down counter direction, but now not available.

Table 5-150: PT_DICounter Member Description

PT_CounterPWMSetting

```
typedef struct tagPT_CounterPWMSetting
{
USHORT        Port;
   FLOAT      Period;
   FLOAT      HiPeriod;
   ULONG      OutCount;
   USHORT     GateMode;
} PT_CounterPWMSetting, FAR * LPT_CounterPWMSetting;
```

PT_CounterPWMSetting is used by **DRV_CounterPWMSetting** function .

Member Description

Name	Direction	Type	Description
Port	Input	unsigned short	counter port
Period	Input	floating-point	time period
HiPeriod	Input	floating-point	upcycle time period
OutCount	Input	long	stop count
GateMode	Input	unsigned short	gate mode of PWM

Table 5-151: PT_CounterPWMSetting Member Description

PT_AlarmConfig

```
typedef struct tagPT_AlarmConfig
{
  USHORT    chan;
  FLOAT     LoLimit;
  FLOAT     HiLimit;
} PT_AlarmConfig, FAR * LPT_AlarmConfig;
```

PT_AlarmConfig is used by **DRV_AlarmConfig** function .

Member Description

Name	Direction	Type	Description
chan	Input	unsigned short	the channel for alarm monitoring
LoLimit	Input	floating point	low limit value for alarm monitoring
HiLimit	Input	floating point	high limit value for alarm monitoring

Table 5-152: PT_AlarmConfig Member Description

PT_AlarmEnable

```
typedef struct tagPT_AlarmEnable
{
  USHORT    chan;
  USHORT    LatchMode;
  USHORT    enabled;
} PT_AlarmEnable, FAR * LPT_AlarmEnable;
```

PT_AlarmEnable is used by **DRV_AlarmEnable** function .

Member Description

Name	Direction	Type	Description
chan	Input	unsigned short	the channel for alarm monitoring
LatchMode	Input	unsigned short	momentary(0), latching(1)
enabled	Input	unsigned short	enable(1), disable(0)

Table 5-153: PT_AlarmEnable Member Description

PT_AlarmCheck

```
typedef struct tagPT_AlarmCheck
{
  USHORT    chan;
  USHORT far  *LoState;
  USHORT far  *HiState;
} PT_AlarmCheck, FAR * LPT_AlarmCheck;
```

PT_AlarmCheck is used by **DRV_AlarmCheck** function .

Member Description

Name	Direction	Type	Description
chan	Input	unsigned short	the channel for alarm monitoring
LoState	Output	long pointer to unsigned short	the current state of the low alarm limit
HiState	Output	long pointer to unsigned short	the current state of the high alarm limit

Table 5-154: PT_AlarmCheck Member Description

PT_FAIIntStart

```
typedef struct tagPT_FAIIntStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    chan;
  USHORT    gain;
  USHORT far  *buffer;
  ULONG     count;
  USHORT    cyclic;
  USHORT    IntrCount;
} PT_FAIIntStart, FAR * LPT_FAIIntStart;
```

PT_FAIIntStart is used by **DRV_FAIIntStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
gain	Input	unsigned short	depends on input range on hardware	gain code
buffer	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer allocated by user
count	Input	unsigned long	1-65536	conversion count
cyclic	Input	unsigned short	0,1	cyclic mode: cyclic (1), non cyclic (0)
IntrCount	Input	Unsigned short	depends on hardware	count to interrupt

Table 5-155: PT_FAIIntStart Member Description

Note

The **IntrCount** value for PCL-818HD and PCL818HG can be set to **1** or **FIFO_SIZE**(512), the **IntrCount** value for PCL-1800 is varied (>=**1**). But the **IntrCount** value for other cards which has no FIFO on board should be set to 1 only.

The **IntrCount** value for PCI-1710 can be set to **1** or **FIFO_SIZE**(2048), the **IntrCount** value for PCI-1710 is varied (>=**1**). But the **IntrCount** value for other cards which has no FIFO on board should be set to 1 only.

PT_FAIIntScanStart

```
typedef struct tagPT_FAIIntScanStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    NumChans;
  USHORT    StartChan;
   USHORT far *GainList;
  USHORT far  *buffer;
  ULONG     count;
  USHORT    cyclic;
  USHORT    IntrCount;
} PT_FAIIntScanStart, FAR * LPT_FAIIntScanStart;
```

PT_FAIIntScanStart is used by **DRV_FAIIntScanStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
NumChans	Input	unsigned short	0-n (n depends on the available channels on hardware)	number of channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	start channel of the scan channel
GainList	Input	long pointer to unsigned short array with **NumChans** entries	depends on input range on hardware	gain code array for the scan channel
buffer	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer allocated by user
count	Input	unsigned long	1-65536	conversion count
cyclic	Input	unsigned short	0,1	cyclic mode: cyclic (1), non cyclic (0)
IntrCount	Input	unsigned short	depends on hardware	count to interrupt

Table 5-156: PT_FAIIntScanStart Member Description

Note

The **IntrCount** value for PCL-818HD and PCL818HG can be set to **1** or **FIFO_SIZE**(512), the **IntrCount** value for PCL-1800 is varied (>=**1**). But the **IntrCount** value for other cards which has no FIFO on board should be set to 1 only.

The **IntrCount** value for PCI-1710 can be set to **1** or **FIFO_SIZE**(2048), the **IntrCount** value for PCI-1710 is varied (>=**1**). But the **IntrCount** value for other cards which has no FIFO on board should be set to 1 only.

PT_FAIDmaStart

```
typedef struct tagPT_FAIDmaStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    chan;
  USHORT    gain;
  USHORT far  *buffer;
  ULONG     count;
} PT_FAIDmaStart, FAR * LPT_FAIDmaStart;
```

PT_FAIDmaStart is used by **DRV_FAIDmaStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
gain	Input	unsigned short	depends on input range on hardware	gain code
buffer	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer
count	Input	unsigned long	1-65536	conversion count

Table 5-157: PT_FAIDmaStart Member Description

PT_FAIDmaScanStart

```
typedef struct tagPT_FAIDmaScanStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    NumChans;
  USHORT    StartChan;
  USHORT far  *GainList;
  USHORT far  *buffer;
  ULONG     count;
} PT_FAIDmaScanStart, FAR * LPT_FAIDmaScanStart;
```

PT_FAIDmaScanStart is used by **DRV_FAIDmaScanStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
NumChans	Input	unsigned short	0-n (n depends on the available channels on hardware)	number of channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	start channel of the scan channel
GainList	Input	long pointer to unsigned short array with **NumChans** entries	depends on input range on hardware	gain code array for the scan channel
buffer	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer allocated by user
count	Input	unsigned long	1-65536	conversion count

Table 5-158: PT_FAIDmaScanStart Member Description

PT_FAIDualDmaStart

```
typedef struct tagPT_FAIDualDmaStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    chan;
  USHORT    gain;
  USHORT far  *BufferA;
  USHORT far  *BufferB;
  ULONG     count;
} PT_FAIDualDmaStart, FAR * LPT_FAIDualDmaStart;
```

PT_FAIDualDmaStart is used by **DRV_FAIDualDmaStart** function
.

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
gain	Input	unsigned short	depends on input range on hardware	gain code
BufferA	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer
BufferB	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer
count	Input	unsigned long	1-65536	conversion count

Table 5-159: PT_FAIDualDmaStart Member Description

PT_FAIDualDmaScanStart

```
typedef struct tagPT_FAIDualDmaScanStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    NumChans;
  USHORT    StartChan;
  USHORT far  *GainList;
  USHORT far  *BufferA;
  USHORT far  *BufferB;
  ULONG     count;
} PT_FAIDualDmaScanStart, FAR * LPT_FAIDualDmaScanStart;
```

PT_FAIDualDmaScanStart is used by **DRV_FAIDualDmaScanStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
NumChans	Input	unsigned short	0-n (n depends on the available channels on hardware)	number of channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	start channel of the scan channel
GainList	Input	long pointer to unsigned short array with **NumChans** entries	depends on input range on hardware	gain code array for the scan channel
BufferA	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer allocated by user
BufferB	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer allocated by user
count	Input	unsigned long	1-65536	conversion count

Table 5-160: PT_FAIDualDmaScanStart Member Description

PT_FAITransfer

```
typedef struct tagPT_FAITransfer
{
  USHORT    ActiveBuf,
  LPVOID    DataBuffer;
  USHORT    DataType;
  ULONG     start;
  ULONG     count;
  USHORT far *overrun;
} PT_FAITransfer, FAR * LPT_FAITransfer;
```

PT_FAITransfer is used by **DRV_FAITransfer** function .

Member Description

Name	Direction	Type	Range	Description
ActiveBuf	Input	unsiged short	0,1	0 - buffer A, (or single buffer), 1- buffer B
DataBuffer	Output	long pointer to floating-point or unsigned short	depends on hardware	data array
DataType	Input	unsigned short	0,1	data type: unsigned short (0), float-point(1)
start	Input	unsigned short	0-65535	start point of the source buffer to be copied to the data buffer
count	Input	unsigned short	1-65535	number of points in the source buffer to be copied to the data buffer
overrun	Output	unsigned short	0,1	overrun status: overrun (1), no overrun (0)

Table 5-161: PT_FAITransfer Member Description

PT_FAICheck

```
typedef struct tagPT_FAICheck
{
  USHORT far  *ActiveBuf;
  USHORT far  *stopped;
   ULONG  far  *retrieved;
  USHORT far  *overrun;
  USHORT far  *HalfReady;
} PT_FAICheck, FAR * LPT_FAICheck;
```

PT_FAICheck is used by **DRV_FAICheck** function .

Member Description

Name	Direction	Type	Range	Description
ActiveBuf	Output	long pointer to unsigned short	0,1	A return value from the function to indicate which buffer is being filled during high speed data transfer. 0 - buffer A(or single buffer), 1- buffer B
stopped	Output	long pointer to unsigned short	0,1	indicates the operation is complete (1), or incomplete (0)
retrieved	Output	long pointer to unsigned short	0-65536	conversion count stored in the buffer
overrun	Output	long pointer to unsigned short	0,1	indicates the data in the buffer is overrun for cyclic mode
HalfReady	Output	long pointer to unsigned short	0,1,2	indicates the data in the half buffer is full , not ready (0), first half (1), second half (2)

Table 5-162: PT_FAICheck Member Description

PT_FAIWatchdogConfig

```
typedef struct tagPT_FAIWatchdogConfig
{
  USHORT    TrigMode;
  USHORT    NumChans;
  USHORT    StartChan;
  USHORT far  *GainList;
  USHORT far  *CondList;
  TRIGLEVEL far *LevelList;
} PT_FAIWatchdogConfig, FAR * LPT_FAIWatchdogConfig;
```

PT_FAIWathchdogConfig is used by **DRV_FAIWatchdogConfig** function .

Member Description

Name	Direction	Type	Range	Description
TrigMode	Input	unsigned short	0,1,2,3	triggering mode
NumChans	Input	unsigned short	0-n (n depends on the available channels on hardware)	number of channels
StartChan	Input	unsigned short	0-n (n depends on hardware)	start channel of the scan channel
GainList	Input	long pointer to unsigned short array with **NumChans** entries	depends on input range on hardware	gain code array for the scan channel
CondList	Input	long pointer to unsigned short array with **NumChans** entries	0,1,2,3,4	condition array to specify the trigger condition for the scan channel
LevelList	Input	long pointer to **TRIGLEVEL** array with **NumChans** entries	depends on input range on hardware	level array to specify the low and high limit for scan channels

Table 5-163: PT_FAIWatchdogConfig Member Description

PT_FAIIntWatchdogStart

```
typedef struct tagPT_FAIIntWatchdogStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT far  *buffer;
  ULONG     BufferSize;
  ULONG     count;
  USHORT    cyclic;
  USHORT    IntrCount;
} PT_FAIIntWatchdogStart, FAR * LPT_FAIIntWatchdogStart;
```

PT_FAIIntWatchdogStart is used by **DRV_FAIIntWatchdogStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
buffer	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer A assigned by ***AllocateDMABuffer*** or user
BufferSize	Input	unsigned long	1-65536	buffer size
count	Input	unsigned short	1-65535	number of points in the source buffer to be copied to the data buffer
cyclic	Input	unsigned short	0,1	cyclic mode: cyclic (1), non cyclic (0)
IntrCount	Input	unsigned short	depends on hardware	count to interrupt

Table 5-164: PT_FAIIntWatchdogStart Member Description

PT_FAIDmaWathchdogStart

```
typedef struct tagPT_FAIDmaWatchdogStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT far  *BufferA;
  USHORT far  *BufferB;
  ULONG     BufferSize;
  ULONG     count;
} PT_FAIDmaWatchdogStart, FAR * LPT_FAIDmaWatchdogStart;
```

PT_FAIDmaWatchdogStart is used by **DRV_FAIDmaWatchdogStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on the pacer on hardware	sampling rate in second
BufferA	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer A assigned by ***AllocateDMABuffer*** or user
BufferB	Output	long pointer to unsigned short array	depends on I/O register format on hardware	data buffer B assigned by ***AllocateDMABuffer*** or user
BufferSize	Input	unsigned long	1-65536	buffer size
count	Input	unsigned short	1-65535	number of points in the source buffer to be copied to the data buffer

Table 5-165: PT_FAIDmaWathchdogStart Member Description

PT_FAIWathchdogCheck

```
  typedef struct tagPT_FAIWatchdogCheck
  {
    USHORT    DataType;
     USHORT far  *ActiveBuf;
    USHORT far  *triggered;
     USHORT far  *TrigChan;
     ULONG far   *TrigIndex;
LPVOID        TrigData;
  } PT_FAIWatchdogCheck, FAR * LPT_FAIWatchdogCheck;
```

PT_FAIWahchdogCheck is used by **DRV_FAIWahtchdogCheck** function .

Member Description

Name	Direction	Type	Range	Description
DataType	Input	Unsigned short	0,1	Indicate raw data (0) and float data (1)
ActiveBuf	Output	long pointer to unsigned short	0,1	A return value from the function to indicate which buffer is being filled during high speed data transfer. 0 - buffer A(or single buffer), 1- buffer B
Triggered	Output	long pointer to unsigned short	0,1	indicates if the watchdog is triggered (1), or not (0)
TrigChan	Output	long pointer to unsigned short	scan channels	triggered channel if **triggered** is 1
TrigIndex	Output	Long pointer to unsigned long	default	Specifies the TrigData value index in buffer that matchs the monitoring condition.
TrigData	Output	long pointer to unsigned float or ; long pointer to unsigned short	0-65535	the location of the data in the buffer triggers the watchdog

Table 5-166: PT_FAIWathchdogCheck Member Description

PT_FAOIntStart

```
typedef struct tagPT_FAOIntStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    chan;
   LONG   far  *buffer;
  ULONG     count;
  USHORT    cyclic;
} PT_FAOIntStart, FAR * LPT_FAOIntStart;
```

PT_FAOIntStart is used by **DRV_FAOIntStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on hardware	sampling rate in second
chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
Buffer	Output	long pointer to unsigned short array	depends on hardware	data buffer allocated by user
Count	Input	unsigned long	1-65536	output count
Cyclic	Input	unsigned short	0,1	cyclic mode: cyclic (1), non cyclic (0)

Table 5-167: PT_FAOIntStart Member Description

PT_FAODmaStart

```
typedef struct tagPT_FAODmaStart
{
  USHORT    TrigSrc;
  DWORD     SampleRate;
  USHORT    chan;
  LONG   far  *buffer;
  ULONG     count;
} PT_FAODmaStart, FAR * LPT_FAODmaStart;
```

PT_FAODmaStart is used by **DRV_FAODmaStart** function .

Member Description

Name	Direction	Type	Range	Description
TrigSrc	Input	unsigned short	0,1	triggering source: external (1), internal (0)
SampleRate	Input	floating point	depends on hardware	sampling rate in second
Chan	Input	unsigned short	0-n (n depends on hardware)	the sampled channel
Buffer	Output	long pointer to unsigned short array	depends on hardware	data buffer allocated by ***AllocateDMABuffer*** or user
Count	Input	unsigned long	1-65536	output count

Table 5-168: PT_FAODmaStart Member Description

PT_FAOScale

```
typedef struct tagPT_FAOScale
{
  USHORT    chan;
  ULONG     count;
  FLOAT far    *VoltArray;
  USHORT far  *BinArray;
} PT_FAOScale, FAR * LPT_FAOScale;
```

PT_FAOScale is used by **DRV_FAOScale** function .

Member Description

Name	Direction	Type	Range	Description
chan	Input	unsigned short	0-65535	Channel number
Count	Input	unsigned long	1-65536	Conversion count
VoltArray	Input	long pointer to floating-point array	the output range of the hardware	input float-point values of data buffer
BinArray	Output	long pointer to unsigned short	default	binary values converted from the voltages

Table 5-169: PT_FAOScale Member Description

PT_FAOLoad

```
 typedef struct tagPT_FAOLoad
 {
   USHORT    ActiveBuf;
   USHORT far  *DataBuffer;
    USHORT    start;
ULONG        count;
 } PT_FAOLoad, FAR * LPT_FAOLoad;
```

PT_FAOLoad is used by **DRV_FAOLoad** function .

Member Description

Name	Direction	Type	Range	Description
ActiveBuf	Input	unsiged short	0,1	0 - buffer A, (or single buffer), 1-buffer B
DataBuffer	Input	long pointer to unsigned short	default	binary data array
start	Input	unsigned short	0-65535	start point of the source buffer to be copied to the data buffer
count	Input	unsigned short	1-65535	number of points to be transferred from the binary data array

Table 5-170: PT_FAOLoad Member Description

PT_FAOCheck

```
typedef struct tagPT_FAOCheck
{
  USHORT far  *ActiveBuf;
  USHORT far  *stopped;
   ULONG  far  *CurrentCount;
  USHORT far  *overrun;
   USHORT far  *HalfReady;
} PT_FAOCheck, FAR * LPT_FAOCheck;
```

PT_FAOCheck is used by **DRV_FAOCheck** function .

Member Description

Name	Direction	Type	Range	Description
ActiveBuf	Output	long pointer to unsigned short	0,1	A return value from the function to indicate which buffer is being filled during high speed data transfer. 0 - buffer A(or single buffer), 1- buffer B
stopped	Output	long pointer to unsigned short	0,1	indicates the operation is complete (1), or incomplete (0)
CurrentCount	Output	long pointer to unsigned short	0-65536	current output count
Overrun	Output	Long pointer to unsigned short	0,1	Indicates the overrun state, overrun(1)
HalfReady	Output	long pointer to unsigned short	0,1,2	indicates the data in the next half buffer is available for new data , not ready (0), first half (1), second half (2)

Table 5-171: PT_FAOCheck Member Description

PT_EnableEvent

```
typedef struct tagPT_EnableEvent
{
  USHORT    EventType;
  USHORT    Enabled;
  USHORT    Count;
} PT_EnableEvent, FAR * LPT_EnableEvent
```

PT_EnableEvent is used by **DRV_EnableEvent** function .

Member Description

Name	Direction	Type	Range	Description
EventType	Input	unsigned short	default	type of event
Enabled	Input	unsigned short	0 or 1	Enabled (1) or Disabled (0)
Count	Input	unsigned short	default	number of interrupt count will be send event

Table 5-172: PT_EnableEvent Member Description

PT_CheckEvent

```
typedef struct tagPT_CheckEvent
{
  USHORT far *EventType;
  DWORD     Milliseconds;
} PT_CheckEvent, FAR * LPT_CheckEvent;
```

PT_CheckEvent is used by **DRV_CheckEvent** function .

Member Description

Name	Direction	Type	Range	Description
EventType	Output	pointer to unsigned short	type of event	Return event type from driver, please reference ***EnableEvent*** parameter.
Milliseconds	Input	DWORD	0-65536	Time-out interval in milliseconds

Table 5-173: PT_CheckEvent Member Description

PT_AllocateDMABuffer

```
typedef struct tagPT_AllocateDMABuffer
{
  USHORT    CyclicMode;
  ULONG     RequestBufSize;
  ULONG far * ActualBufSize;
  LONG  far * buffer;
} PT_AllocateDMABuffer, FAR * LPT_AllocateDMABuffer;
```

PT_AlarmCheck is used by **DRV_AlarmCheck** function .

Member Description

Name	Direction	Type	Range	Description
CyclicMode	Input	unsigned short	default	non-cyclic(0) and cyclic(1)
RequestBufSize	Input	unsigned long	default	request buffer size in byte
ActualBufSize	Output	pointer to unsigned long	default	actual size can be allocated buffer size in byte
Buffer	Output	pointer to long	default	buffer address

Table 5-174: PT_AllocateDMABuffer Member Description

PT_EnableEventEx

```
  typedef union tagPT_EnableEventEx  //union type struct
  {
PT_DIFilter          Filter;
     PT_DIPattern Pattern;
     PT_DICounter                  Counter;
     PT_DIStatus  Status;
  } PT_EnableEventEx, FAR * LPT_EnableEventEx;
```

PT_EnableEventE*x* is used by **DRV_EnableEventEx** function .

Member Description

Name	Direction	Type	Range	Description
PT_DIFilter	Input	structure pointer	default	Structure for Digin Filter
PT_DIPattern	Input	structure pointer	default	Structure for Pattern Match
PT_DICounter	Input	structure pointer	default	Structure for Counter Match
PT_DIStatus	Input	structure pointer	default	Structure for Change of Input State

Table 5-175: PT_EnableEventEx Member Description

PT_DIFilter

```
typedef struct tagPT_DIFilter
{
USHORT          usEventType;
      USHORT    usEventEnable;
      USHORT    usCount;
      USHORT    usEnable;
      USHORT far * usHiValue;
      USHORT far * usLowValue;
} PT_DIFilter, FAR * LPT_DIFilter;
```

PT_DIFilter is used by **DRV_EnableEventEx** function .

Member Description

Name	Direction	Type	Range	Description
usEventType	Input	unsigned short	default	event type
usEventEnable	Input	unsigned short	default	event enable/disable bit
usCount	Input	unsigned short	default	event count
usEnable	Input	unsigned short	default	counter0 ~ counter7 enable/disable setting. For example, if bit0 = 1, counter0 is enabled.
usHiValue	Input	long pointer to unsigned short	default	record Filter high status value array pointer
usLowValue	Input	long pointer to unsigned short	default	record Filter low status value array pointer

Table 5-176: PT_DIFilter Member Description

PT_DIPattern

```
typedef struct tagPT_DIPattern
{
    USHORT      usEventType;
    USHORT      usEventEnable;
    USHORT      usCount;
    USHORT      usEnable;
    USHORT      usValue;
} PT_DIPattern, FAR * LPT_DIPattern;
```

PT_DIPattern is used by **DRV_EnableEventEx** function .

Member Description

Name	Direction	Type	Range	Description
usEventType	Input	unsigned short	default	event type
usEventEnable	Input	unsigned short	default	counter0 ~ counter7 enable/disable setting. For example, if bit0 = 1, counter0 is enabled.
usCount	Input	unsigned short	default	event count
usEnable	Input	unsigned short	default	the match function of pattern enable/disable setting. For example, if bit0 = 1, the match function of pattern is enabled.
usValue	Input	unsigned short	default	pattern match pre-setting value

Table 5-177: PT_DIPattern Member Description

PT_DIStatus

```
 typedef struct tagPT_Status
 {
USHORT        usEventType;
   USHORT     usEventEnable;
   USHORT     usCount;
   USHORT     usEnable;
   USHORT     usRisingedge;
   USHORT     usFallingedge;
 } PT_DIStatus, FAR * LPT_DIStatus;
```

PT_DIStatus is used by **DRV_EnableEventEx** function .

Member Description

Name	Direction	Type	Range	Description
usEventType	Input	unsigned short	default	event type
usEventEnable	Input	unsigned short	default	status event enable bit
usCount	Input	unsigned short	default	event count
usEnable	Input	unsigned short	default	status change enable data
usRisingedge	Input	unsigned short	default	record Rising edge trigger type
usFallingedge	Input	unsigned short	default	record Falling edge trigger type

Table 5-178: PT_DIStatus Member Description

PT_FDITransfer

```
typedef struct tagPT_FDITransfer
{
    USHORT      usEventType;
    ULONG far *  ulRetData;
} PT_FDITransfer, FAR * LPT_FDITransfer;
```

PT_FDITransfer is used by **DRV_EnableEventEx** function .

Member Description

Name	Direction	Type	Range	Description
usEventType	Input	Unsigned short	default	event type for DI transfer
ulRetData	Output	Unsigned long pointer	default	transfer data

Table 5-179: PT_FDITransfer Member Description

APPENDIX A

DLL Driver Error Codes

This section lists the status codes returned by these driver functions, including the name and description.

Each driver function returns a status code that indicates whether the function was performed successfully. When a function returns a code that is not zero, it means that the function performed failed. You can pass the error code to **DRV_GetErrorMessage** function to return its error message.

The status code is 32-bit. Its format is described in Figure A-1.

Status code (32-bit)

Status code (32-bit)		
Bit 31-28	Bit 27-16	Bit 15-0
serial port used	base address occupied	Error code

Table A-1: Status Code Format

A summary of the status codes is listed in Table A-2.

Error Code	**Error ID**	**Description (Error Message)**
1	MemoryAllocateFailed (*)	Not Enough Memory
2	ConfigDataLost (*)	Configuration Data Lost
3	InvalidDeviceHandle (*)	Invalid Device Handle
4	AIConversionFailed	Analog Input Failure On I/O=%XH
5	AIScaleFailed	Invalid Scaled Value On I/O=%XH
6	SectionNotSupported	Section Not Supported On I/O=%XH
7	**InvalidChannel**	Invalid Channel On I/O=%XH
8	InvalidGain	Invalid Gain Code On I/O=%XH
9	DataNotReady	Data Not Ready On I/O=%XH
10	InvalidInputParam	Invalid Input Parameter On I/O=%XH
11	NoExpansionBoardConfig	No Expansion Board Configuration in Registry/Configuration File On I/O=%XH
12	InvalidAnalogOutValue	Invalid Analog Output Value On I/O=%XH
13	ConfigIoPortFailed	Configure DIO Port Failure On I/O=%XH
14	CommOpenFailed	Open COM %d Failure
15	CommTransmitFailed	Unable to Transmit to COM %d Address %XH
16	CommReadFailed	Unable to Receive from COM %d Address %XH
17	CommReceiveFailed	Invalid Data Received from COM %d Address %XH
18	CommConfigFailed	Configure Communication Port Falied on COM %d
19	CommChecksumError	Checksum Error from COM %d Address %XH
20	InitError	Initialization Failure On I/O=%XH
21	DMABufAllocFailed (*)	No Buffer Allocated for DMA
22	IllegalSpeed	The Sample Rate Exceeds the Upper Limit On I/O=%XH
23	ChanConflict	Background Operation Is Still Running On I/O=%XH
24	**BoardIDNotSupported**	Board ID Is Not Supported On I/O=%XH
25	FreqMeasurementFailed	Time Interval For Frequency Measurement Is Too Small On I/O=%XH
26	CreateFileFailed (*)	Call CreateFile() Failed
27	FunctionNotSupported (*)	Function Not Supported
28	LoadLibraryFailed (*)	Load Library Failed
29	GetProcAddressFailed (*)	Call GetProcAddress() Failed
30	**InvalidDriverHandle** (*)	Invalid Driver Handle
31	InvalidModuleType	Module Type Not Existence On I/O=%XH
32	InvalidInputRange	The Value is Out of Range On I/O=%XH
33	InvalidWindowsHandle	Invalid Windows Handle of Destination on I/O=%XH
34	InvalidCountNumber	Invalid Numver of Conversion On I/O=%XH
35	InvalidInterruptCount	Invalid Number of Interrupt Count On I/O=%XH
36	InvalidEventCount	Invalid Number of Event Count On I/O=%XH
37	OpenEventFailed	Create or Open Event Failed On I/O=%XH
38	InterruptProcessFailed	Interrupt Process Failed On I/O=%XH
39	InvalidDOSetting	Invalid digital output direction setting COM %d Address %XH
40	InvalidEventType	Invalid Event Type On I/O=%XH
41	EventTimeOut	The Time-out Interval Elapsed in Milliseconds Parameter On I/O=%XH

Table A-2: Status Code Summary

Note: * means that the status code only includes error code.

Error Code	Error ID	Description (Error Message)
100	KeInvalidHandleValue	An error occured while starting the device
101	KeFileNotFound	The device has not been created
102	KeInvalidHandle	The handle passed to the function is not a valid
103	KeTooManyCmds	The logic commands have created an apparent endless loop
104	KeInvalidParameter	Passed to the driver contains an invalid parameter
105	KeNoAccess	Attempts to access a port which has not been defined in DEVINST
106	KeUnsuccessful	The operation was not successful
107	KeConInterruptFailure	The driver connects interrupt failure on I/O=%XH
108	KeCreateNoteFailure	The driver creates notification event failure On I/O=%XH
109	KeInsufficientResources	The system resource is insufficient OnI/O=%XH
110	KeHalGetAdapterFailure	An adapter object could not be created On I/O=%XH
111	KeOpenEventFailure	The driver opens notification event failure On I/O=%XH
112	KeAllocCommBufFailure	Allocate DMA buffer failure On I/O=%XH
113	KeAllocMdlFailure	Allocate MDL for DMA buffer failure On I/O=%XH
114	KeBufferSizeTooSmall	The buffer of requisition must be bigger that PAGE_SIZE On I/O=%XH

Table A-3: Status Code Summary

Error Code	Error ID	Description (Error Message)
201	DNInitFailed	DeviceNet Initialization Failed
202	DNSendMsgFailed	Send Message Failed On Port %d MACID %XH
203	DNRunOutOfMsgID	Run Out of Message ID
204	DNInvalidInputParam	Invalid Input Parameters
205	DNErrorResponse	Error Response On Port %d MACID %XH
206	DNNoResponse	No Response On Port %d MACID %XH
207	DNBusyOnNetwork	Busy On Network On Port %d MACID %XH
208	DNUnknownResponse	Unknown Response On Port %d MACID %XH
209	DNNotEnoughBuffer	Message Length Is Too Long on Port %d MACID %XH
210	DNFragResponseError	Fragment Response Error On Port %d MACID %XH
211	DNTooMuchDataAck	Too Much Data Acknowledge On Port %d MACID %XH
212	DNFragRequestError	Fragment Request Error On Port %d MACID %XH
213	DNEnableEventError	Event Enable/Disable Error On Port %d MACID %XH
214	DNCreateOrOpenEventError	Device Net Driver Create/Open Event Failed On Port %d MACID %XH
215	DNIORequestError	IO Message Request Error On Port %d MACID %XH
216	DNGetEventNameError	Get Event Name From CAN Driver Failed On Port %d MACID %XH
217	DNTimeOutError	Wait For Message Time Out Error On Port %d MACID %XH
218	DNOpenFailed	Open CAN Card Failed
219	DNCloseFailed	Close CAN Card Failed
220	DNResetFailed	DeviceNet Reset Failed

Table A-4: Status Code Summary